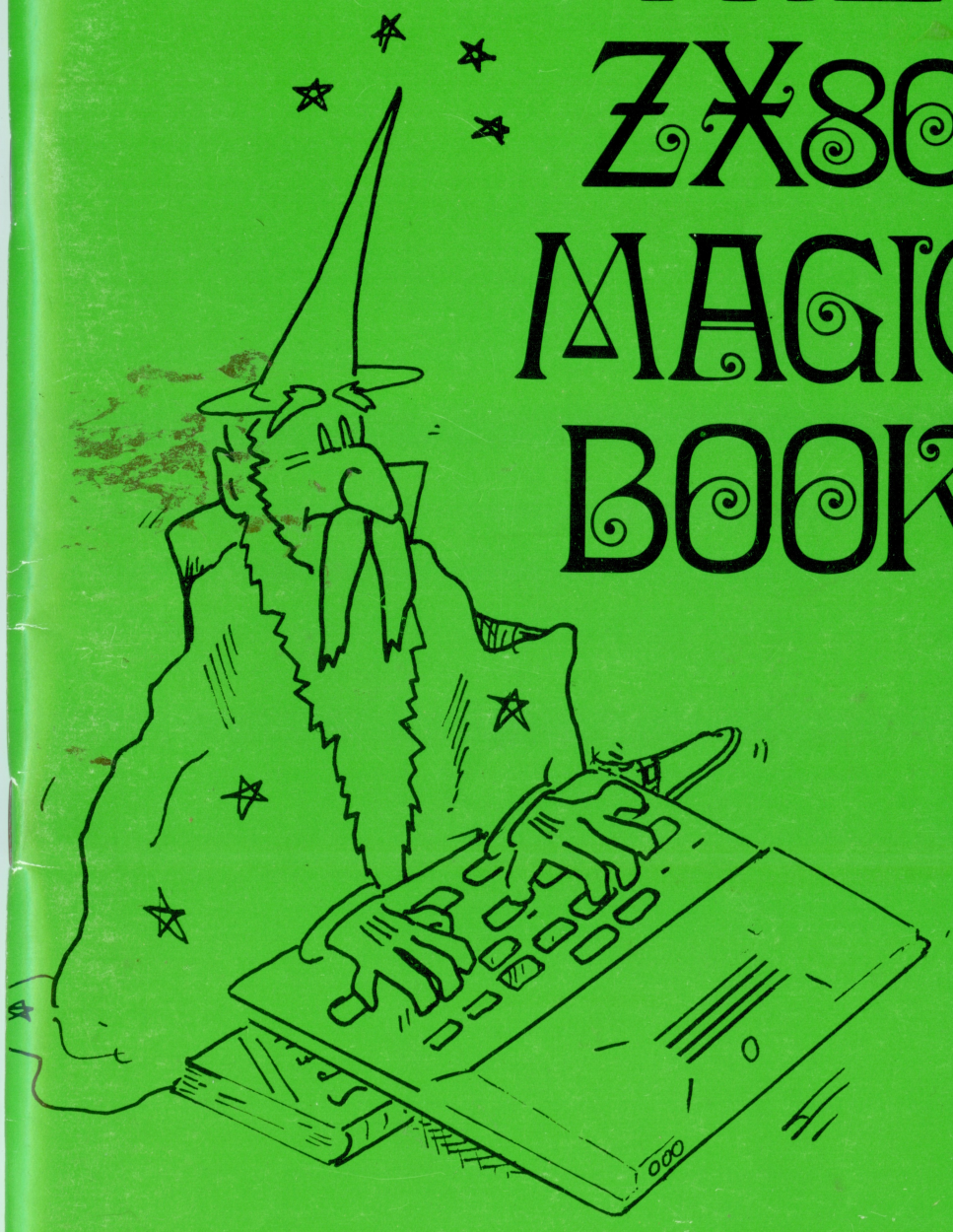
WITH ZX81
SUPPLEMENT

THE ZX80 MAGIC BOOK



THE ZX80 MAGIC BOOK

For the Integer (4K ROM) ZX80 with 1-4K RAM

Contents

PAGE	
2	Russian Roulette
3	Matches
4	Number Guessing A & B
5	Dice(s)
6	Horse Race
7	Buzz-Word
8	Moon Lander
9	Higher-Lower
10	Hammurabi
12	Nibbler, Measles
13	RND
14	Hexpawn
17	Othello
20	Decimal Peeker
21	Hex Peeker
22	Animals
25	Sums Tester
26	More Sums
27	Weekday
28	Music
33	Plotting
35	Creating a Program
40	Assorted Tips
41	Debugging
42	Converting Other BASICs
48	How it Works
52	Using USR
54	Micromon
56	Improving the Picture
58	Memory Map
59	Adding Memory & I/O

© TIMEDATA Ltd.

Edition 1 ; Jun 1980
2 ; Sep 1980
3 ; Oct 1980
Reprinted ; Feb 1981

Russian Roulette

Is a game for two or more players who take turns to place a revolver loaded with just one bullet to their heads and pull the trigger. The game is over when one player refuses to take his turn, or when the revolver fires.

To pull the trigger, press NEWLINE. To quit, press **N** then NEWLINE.

```
  1 REM RUSSIAN ROULETTE
  5 RANDOMISE
100 PRINT "THIS IS RUSSIAN ROULETTE"
110 PRINT
120 PRINT "YOU HAVE A REVOLVER"
130 PRINT "IT HAS 5 EMPTY CHAMBERS"
140 PRINT "AND ONE LOADED ONE"
150 PRINT
160 PRINT "DO YOU WANT TO PULL THE TRIGGER ?"
170 INPUT I$
180 CLS
190 IF CODE(I$)=CODE("N") THEN GO TO 500
200 FOR I=1 TO 200
210 NEXT I
300 IF RND(6)=1 THEN GO TO 400
310 PRINT "----CLICK----"
320 PRINT
330 PRINT
340 PRINT
350 PRINT
360 PRINT
370 GO TO 150
400 PRINT "BANG - YOU ARE DEAD"
410 PRINT
420 PRINT ,CHR$(3)
430 PRINT ,CHR$(2);CHR$(0);CHR$(130)
440 PRINT ,CHR$(2);CHR$(0);CHR$(130)
450 PRINT ,CHR$(128);CHR$(128);CHR$(128);CHR$(3);CHR$(128)
460 STOP
500 PRINT ,"CHICKEN"
```


Matches

I don't want to set the world on fire.

Matches is a simplified form of NIM. Starting with a single row of between 21 and 32 matches, the player and the computer take it in turns to remove 1,2 or 3 matches. The one left with the last match is the loser.

```
 10 REM MATCHES
 20 RANDOMISE
100 LET M = 20 + RND(12)
110 PRINT "WE TAKE TURNS TO REMOVE"
120 PRINT "1,2 OR 3 MATCHES"
130 PRINT "THE ONE LEFT WITH THE LAST"
140 PRINT "MATCH LOSES"
150 PRINT
200 FOR L = 1 TO 4
210 FOR K = 1 TO M
220 IF L = 1 THEN PRINT "□";
225 IF L > 1 THEN PRINT "□";
230 NEXT K
240 PRINT
250 NEXT L
260 PRINT
300 PRINT M;" MATCHES, YOUR TURN"
310 IF M=1 THEN GO TO 510
320 PRINT "HOW MANY WILL YOU TAKE ?"
330 INPUT T
340 IF T = M THEN GO TO 500

350 IF T>3 OR T<1 OR T>M THEN GO TO 330
360 LET M = M - T
370 IF M = 1 THEN GO TO 600
400 LET R = M - 4 * (M/4)
410 IF R = 1 THEN LET C = RND(3)
420 IF NOT R = 1 THEN LET C = R + 3 - 4 * ((R+3)/4)
430 LET M = M - C
440 IF M = 0 THEN GO TO 600
450 CLS
460 PRINT "YOU TOOK ";T
470 PRINT "SO I TOOK ";C
480 PRINT
490 GO TO 200
500 CLS
510 PRINT "I WON"
520 STOP
600 CLS
610 PRINT "YOU WON"
```

Number Guessing A & B

```
 10 REM NUMBER GUESSING A
 20 RANDOMISE
 30 LET N=RND(9)
 40 CLS
 50 PRINT "I AM THINKING OF"
 60 PRINT "A NUMBER FROM 1 TO 9"
 70 PRINT " YOU HAVE 5 GUESSES"
 80 PRINT
 90 FOR K = 1 TO 5
100 INPUT I
110 PRINT I;" IS ";
120 IF I = N THEN GO TO 200
130 PRINT "WRONG"
140 NEXT K
150 PRINT
160 PRINT "IT WAS ";N
170 STOP
200 PRINT "CORRECT"
```

?

In both of the games on this page the computer picks a random number which the player tries to guess.

Game A gives the player 5 chances to guess the number, which is in the range 1 to 9. Unless the player has considerable ESP talent, he should only be able to win about half of the games played.

In contrast, game B allows the player 7 chances to guess a number from 1 to 99, but tells the player whether each (unsuccessful) guess is too high or too low; which is enough information to allow the correct number to be deduced every game.

?

```
 10 REM NUMBER GUESSING B
 20 RANDOMISE
 30 LET N=RND(99)
 40 CLS
 50 PRINT "I AM THINKING OF"
 60 PRINT "A NUMBER FROM 1 TO 99"
 70 PRINT "YOU HAVE 7 GUESSES"
 80 PRINT
 90 FOR K=1 TO 7
100 INPUT I
110 PRINT I;" IS ";
120 IF I=N THEN GO TO 200
130 IF I<N THEN PRINT "TOO SMALL"
140 IF I>N THEN PRINT "TOO BIG"
150 NEXT K
160 PRINT
170 PRINT "IT WAS ";N
180 STOP
200 PRINT "CORRECT"
```


Dice(s)

```
  1 REM DICE THROWER
  5 RANDOMISE
 10 LET N=RND(6)
 20 FOR L=1 TO 9
 30 PRINT "      ";
 40 GO SUB 1000
 50 PRINT
 60 NEXT L
 62 PRINT
 63 PRINT
 65 PRINT "NEWLINE TO THROW"
 70 INPUT A$
 80 CLS
 90 IF A$="" THEN GO TO 10
100 STOP
1000 IF L>1 AND L<9 THEN PRINT CHR$(130);
1010 GO TO 1020+5*L
1025 PRINT "▛▀▀▀▀▀▀▀▜";
1026 RETURN
1030 PRINT "       ▌";
1031 RETURN
1035 IF N<3 THEN GO TO 1070
1036 IF N=3 THEN GO TO 1075
1037 GO TO 1080
1040 GO TO 1030
1045 IF N=(N/2)*2+1 THEN GO TO 1085
1046 IF N=4 THEN GO TO 1030
1047 GO TO 1080
1050 GO TO 1030
1055 IF N<3 THEN GO TO 1070
1056 IF N=3 THEN GO TO 1090
1057 GO TO 1080
1060 GO TO 1030
1065 LET L$=CHR$(131)
1066 PRINT "▌";L$;L$;L$;L$;L$;L$;L$;"▐";
1067 RETURN
1070 PRINT "       ▌";
1071 RETURN
1075 PRINT "     ■ ▌";
1076 RETURN
1080 PRINT " ■   ■ ▌";
1081 RETURN
1085 PRINT "   ■   ▌";
1086 RETURN
1090 PRINT " ■     ▌";
1091 RETURN
```

To throw two die at once, change lines 10 - 55 to;

```
 10 LET N1=RND(6)
 15 LET N2=RND(6)
 20 FOR L=1 TO 9
 25 LET N=N1
 30 PRINT "   ";
 35 GOSUB 1000
 40 LET N=N2
 45 PRINT "   ";
 50 GOSUB 1000
 55 PRINT
```

Horse Race

This program displays two horses, which advance towards the finishing line every time NEWLINE is pressed.

The main body of the program is lines 500 - 640 which display the track, then wait for NEWLINE to be pressed before calculating the new positions.
Lines 1000 - 1110 are a subroutine which displays one horse and the section of the finishing line directly ahead of him. Lines 2000 - 2060 display the section of track and finishing line inbetween the two horses.

Their rate of progress can be changed by altering lines 620 and 630, and artistically minded programmers may care to experiment with lines 1030 - 1060 to achieve a more pleasing representation of a horse and rider.

```
  1 REM HORSE RACE
100 RANDOMISE
110 LET N1=1
120 LET N2=1
500 CLS
510 LET N=N1
520 GO SUB 1000
530 GO SUB 2000
540 LET N=N2
550 GO SUB 1000
560 PRINT
570 IF N1>27 OR N2>27 THEN GO TO 700
600 PRINT "NEWLINE TO MOVE"
610 INPUT A$
620 LET N1=N1+RND(5)
630 LET N2=N2+RND(5)
640 GO TO 500
700 PRINT "END OF RACE"
710 PRINT "NEWLINE FOR ANOTHER"
720 INPUT A$
730 IF A$="" THEN RUN
740 STOP
1000 IF N>28 THEN LET N=28
1005 FOR R=1 TO 4
1010 FOR I=1 TO 28
1020 IF NOT (I=N) THEN GO TO 1070
1030 IF R=1 THEN PRINT " ▣ ";
1040 IF R=2 THEN PRINT " ";CHR$(135);CHR$(133);
1050 IF R=3 THEN PRINT CHR$(128);CHR$(128);"▣";
1060 IF R=4 THEN PRINT "▣";CHR$(130);" ";
1070 IF NOT (I=N) THEN PRINT " ";
1080 NEXT I
1090 PRINT "▣"
1100 NEXT R
1110 RETURN
2000 FOR R=1 TO 3
2010 FOR I=1 TO 30
2020 PRINT " ";
2030 NEXT I
2040 PRINT "▣"
2050 NEXT R
2060 RETURN
```


Buzz – Word

An aid for the busy engineer; BUZZ-WORD generates impressive sounding phrases for use in reports to higher management.

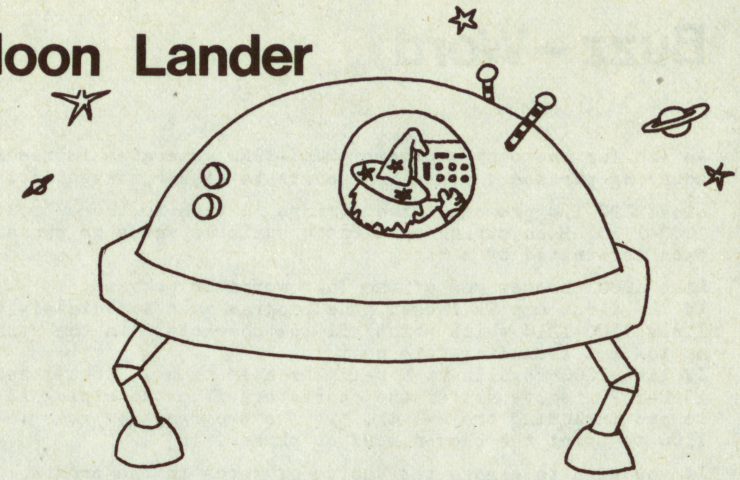
Lines 120-170 present three strings in turn to the subroutine 1000-1210. Each string contains 4 suitable words or phrases each terminated by a *.

Line 1000 chooses one of the four words or phrases.
If the first one is chosen, the program goes immediately to lines 1100-1210 which print all the characters in the string up to- but excluding- the next *.
If line 1000 results in N being greater than 1, then lines 1020-1070 remove all of the characters from the string A$ up to and including the N-1 th. *. The program then goes to line 1100 to print the chosen word or phrase.

If you wish to change the choice of words in the program;
- Lines 120, 140 and 160 must each contain an equal number of words or phrases, each terminated by *.
- Line 1000 must reflect the number of words or phrases in lines 120, 140 and 160.
- Each word or phrase should be no longer than 10 characters to prevent the resulting print-out being longer than one display line.

```
 10 REM BUZZ-WORD
100 CLS
110 RANDOMISE
120 LET A$ = "GLOBAL*PARALLEL*FEASIBLE*LATERAL*"
130 GOSUB 1000
140 LET A$ = "SILICON*TOP DOWN* PRAGMATIC*MODULAR*"
150 GOSUB 1000
160 LET A$ = "CONCEPT*SCENARIO*PROJECTION*PHASE*"
170 GOSUB 1000
180 PRINT
190 PRINT
200 PRINT "HIT NEWLINE FOR ANOTHER PHRASE"
210 INPUT A$
220 IF A$ = "" THEN GOTO 100
230 STOP
1000 LET N=RND(4)
1010 IF N = 1 THEN GO TO 1100
1020 IF CODE(A$) = CODE("*") THEN GO TO 1050
1030 LET A$ = TL$(A$)
1040 GO TO 1020
1050 LET A$ = TL$(A$)
1060 LET N = N - 1
1070 GO TO 1010
1100 IF CODE(A$) = CODE("*") THEN GOTO 1200
1110 PRINT CHR$(CODE(A$));
1120 LET A$ = TL$(A$)
1130 GO TO 1100
1200 PRINT " ";
1210 RETURN
```

Moon Lander



You are piloting a rocket landing on the Moon. (The ***!!*** computer has failed again !!). At the start of the simulation you are 500 feet up, falling at 75 feet per second, and have 125 units of fuel. Every second you are given a display of your height, speed, and remaining fuel. You then have to decide how much fuel to burn during the next second to bring you to a safe landing. If the speed shown is negative, then you are travelling away from the surface of the moon.

The program is straightforward, except for lines 300 - 360, which approximate the complex equations needed for a proper calculation of your trajectory by using a simple formula repeated at 1/10 second intervals.

```
  1 REM MOON LANDER
100 LET H=5000
110 LET V=750
120 LET F=125
200 CLS
210 PRINT "HEIGHT SPEED   FUEL   BURN"
220 PRINT
230 FOR K=1 TO 15
240 PRINT H/10,V/10,F,
250 IF F>0 THEN INPUT R
260 IF R>F THEN LET R=F
270 PRINT R
300 FOR T=1 TO 10
310 LET H=H-V/10-(5-R)/2
320 IF H<1 THEN GO TO 500
330 LET V=V+5-R
340 NEXT T
350 LET F=F-R
360 NEXT K
370 GO TO 200
500 LET F=F-R*T/10
510 LET V=ABS(V)
520 IF V>50 THEN PRINT "YOU CRASHED"
530 IF V<51 AND V>20 THEN PRINT "A BUMPY LANDING"
540 IF V<21 THEN PRINT "A GOOD LANDING"
550 PRINT "AT ";V/10;" FEET/SEC"
560 PRINT "WITH ";F;" FUEL LEFT"
```


Higher Lower

Higher/Lower is played by one person betting against the computer. He is given a starting stake of £ 100.

For each play, the computer picks two numbers between 1 and 99, displays the first, then asks the player to bet any or all of his money on whether the second number is higher or lower than the first. A really successful player can break the bank by amassing £ 10,000 or more. The game will also stop when the player has no money left, or if he bets £ 0.

```
 10 REM HIGHER/LOWER
100 LET P=100
110 RANDOMISE
120 LET N1=RND(99)
130 LET N2=RND(99)
140 CLS
150 PRINT "I HAVE CHOSEN 2 NUMBERS (1-99)"
160 PRINT "THE FIRST IS ";N1
170 PRINT
180 PRINT "IS THE SECOND NUMBER"
190 PRINT "HIGHER (H) OR LOWER (L) ?"
200 INPUT I$
210 IF I$="" THEN STOP
220 IF I$="H" THEN GO TO 250
230 IF I$="L" THEN GO TO 250
240 GO TO 200
250 CLS
260 PRINT "YOU THINK IT IS ";
270 IF I$="H" THEN PRINT "HIGHER";
280 IF I$="L" THEN PRINT "LOWER";
290 PRINT " THAN ";N1
300 PRINT "YOU HAVE £";P
310 PRINT "HOW MUCH DO YOU BET ?"
320 INPUT B
330 IF B=0 THEN STOP
340 IF B>P THEN GO TO 320
350 IF N2=N1 THEN GO TO 400
360 LET P=P-B
370 IF N2>N1 AND I$="L" THEN GO TO 400
380 IF N2<N1 AND I$="H" THEN GO TO 400
390 LET P=P+2*B
400 CLS
410 PRINT "IT WAS ";N2;" YOU NOW HAVE £";P
420 IF P>9999 THEN GO TO 500
430 IF P=0 THEN GO TO 600
440 PRINT "HIT NEWLINE FOR ANOTHER BET"
450 INPUT I$
460 IF I$="" THEN GO TO 120
470 STOP
500 PRINT
510 PRINT "CONGRATULATIONS - YOU HAVE"
520 PRINT "BROKEN THE BANK"
530 STOP
600 PRINT
610 PRINT "SORRY"
```

Hammurabi

This program needs 2K of memory.

You play the part of Hammurabi; king of ancient Sumaria. Each year the computer tells you how big your kingdom is, how many subjects you have, and how much grain there is as a result of the harvest and that stored from the previous year. You must then decide whether to trade land for grain with your neighbouring kingdoms, how much to allocate as food for the coming year, and how much grain to use as seed.

Your subjects will die off if they don't get sufficient food (about 15 bushels per head per year); if you give them too much food then the surplus is wasted. Each year's harvest depends on the amount of land available, the amount of grain allocated as seed, the number of people to work the land, and a large random factor (the weather ?). Your kingdom is also subject to the plague about every sixth year, and rats can devastate your valuable grain stores.

Hammurabi is one of the original computer games; the author can remember seeing it played in the mid 1960's. As set up here it is extremely difficult to prevent your kingdom from collapsing around you; if you wish to change the parameters then;

R is the number of bushels eaten by rats.
C, the harvest, is affected by lines 640 & 650 as well as 710.
D is the number of deaths from starvation.
N is the number of births.
T is the number of deaths from the plague.

```
  1 REM HAMMURABI
100 RANDOMISE
110 LET H=100
120 LET S=3000
130 LET A=1000
140 LET Y=0
150 LET A$="O HAMMURABI, KING OF SUMERIA"
200 PRINT A$
210 PRINT
220 PRINT "YOUR POPULATION IS ";H
230 PRINT "YOU HAVE ";S;" BUSHELS OF GRAIN"
240 PRINT "YOU HAVE ";A;" ACRES OF LAND"
250 PRINT
300 LET L=16+RND(8)
310 PRINT "LAND IS ";L;" BUSHELS PER ACRE"
320 PRINT "HOW MANY ACRES WILL YOU BUY ";
330 INPUT AA
340 IF AA*L > S THEN GO TO 330
350 PRINT AA
400 IF AA > 0 THEN GO TO 460
410 PRINT "HOW MANY ACRES WILL YOU SELL ";
420 INPUT AA
430 IF AA > A THEN GO TO 420
440 PRINT AA
450 LET AA=-AA
460 LET A=A+AA
470 LET S=S-L*AA
500 PRINT "HOW MANY BUSHELS FOR FOOD ";
510 INPUT F
520 IF F > S THEN GO TO 510
530 PRINT F
540 LET S=S-F
600 PRINT "HOW MANY BUSHELS FOR SEED"
610 INPUT P
620 IF P > S THEN GO TO 610
630 LET S=S-P
```

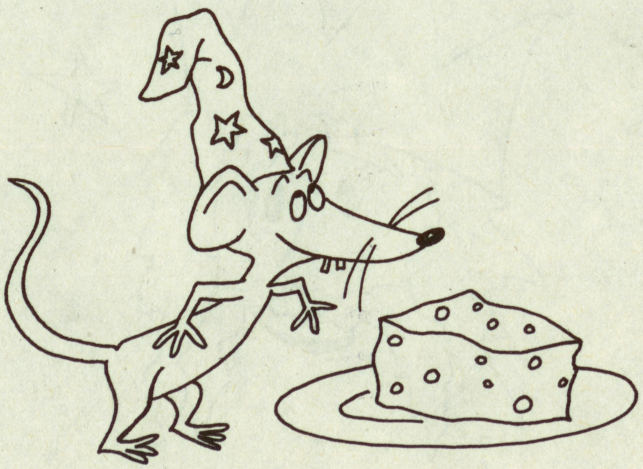


```
640 IF P > 2*A THEN LET P=2*A
650 IF P > 25*H THEN LET P=25*H

700 LET R=RND(S/2)-1
710 LET C=RND(4*P)
720 LET S=S+C-R
730 LET D=H-F/15
740 IF D < 1 THEN LET D=0
750 LET N=H/10+RND(H/20)
760 LET H=H-D+N
770 LET T=0
780 IF RND(6)=3 THEN LET T=H/8+RND(H/3)
790 LET H=H-T
800 CLS
810 PRINT A$
815 PRINT
820 LET Y=Y+1
825 PRINT "YEAR ";Y;" OF YOUR REIGN"
830 IF D>H/4 OR R>S/4 OR T>H/4  OR C<A THEN
    PRINT "WAS A BAD YEAR"
840 IF D=0 AND R<S/10 AND T=0 AND C>3*A THEN
    PRINT "WAS A GOOD YEAR"
845 PRINT
850 PRINT "RATS ATE ";R;" BUSHELS"
860 PRINT "YOU HARVESTED ";C;" BUSHELS"
870 PRINT N;" BABIES WERE BORN"
880 PRINT D;" PEOPLE STARVED"
885 IF T>0 THEN PRINT T;" DIED OF PLAGUE"
890 IF H>0 THEN GO TO 210
900 PRINT "THAT IS THE END OF YOUR REIGN"
```

Nibbler

Nibbler is a shorter version of the 'Cheese Nibbler' program in the ZX80 book. In this program the mouse gets a lump of cheese every time. As with some other programs in this book, it asks you to press newline to continue; and the program stops if you enter any character (if line 300 gets anything other than a null string).

```
 10 REM CHEESE NIBBLER
100 DIM A(30)
110 RANDOMISE
140 LET P$=" THIS IS THE CHEESE."
150 LET Q$=" HIT NEWLINE TO LET"
160 LET R$=" THE MOUSE NIBBLE IT"
200 CLS
210 FOR J=1 TO 3
220 FOR K=1 TO 10
230 LET B=K+10*(J-1)
240 IF A(B)=0 THEN PRINT "▦";
250 IF A(B)=1 THEN PRINT " ";
260 NEXT K
265 IF J=1 THEN PRINT P$
270 IF J=2 THEN PRINT Q$
275 IF J=3 THEN PRINT R$
280 NEXT J
300 INPUT I$
310 IF NOT I$="" THEN STOP
320 LET B=RND(30)
330 IF A(B)=1 THEN GO TO 320
340 LET A(B)=1
350 GO TO 200
```



RND

So many programs depend on the RND function, that it is worth checking to see how random it is.

This program does a number of RND(20) operations, then displays a histogram of the result. The histogram has 20 bars, corresponding to the numbers 1 - 20 , the length of each bar being proportional to the frequency with which RND(20) came up with that number.

When the program has run, and the histogram is displayed, pressing NEWLINE will run it again, adding the new values to the previous ones.

The 'scores' are stored in the array A(20) by lines 110 - 140. Lines 200-290 display the 20 line histogram, with line 245 ensuring that each bar cannot overflow the end of the screen.

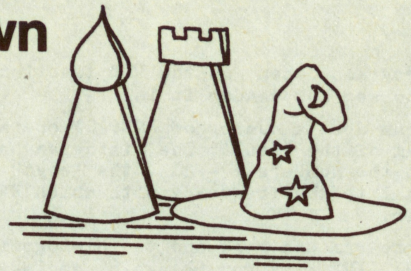
```
 10 REM RND FUNCTION
 20 DIM A(20)
100 RANDOMISE
110 FOR K=1 TO 50
120 LET N=RND(20)
130 LET A(N)=A(N)+1
140 NEXT K
200 CLS
210 FOR K=1 TO 20
220 IF K< 10 THEN PRINT " ";
230 PRINT K;" ";
240 IF A(K) < 1 THEN GO TO 280
245 IF A(K) > 29 THEN LET A(K)=29
250 FOR P=1 TO A(K)
260 PRINT "▦";
270 NEXT P
280 PRINT
290 NEXT K
300 PRINT "NEWLINE FOR MORE"
310 INPUT A$
320 IF A$="" THEN GO TO 100
```

Measles

Measles paints the screen with randomly placed dots. It is a trivial program, but pleasing to those of a certain age,

```
 10 REM MEASLES
100 DIM A(128)
200 PRINT "HOW MANY SPOTS (1-100) ?"
210 INPUT K
220 IF K>100 THEN GO TO 210
230 FOR J=1 TO K
240 LET B=RND(128)
250 IF A(B)=1 THEN GO TO 240
260 LET A(B)=1
270 NEXT J
300 CLS
310 FOR J=1 TO 192
320 IF A(J)=0 THEN PRINT "   ";
330 IF A(J)=1 THEN PRINT "▣  ";
340 NEXT J
```

Hexpawn



This program requires 3K of RAM.

Hexpawn is a simple board game played between the computer and a human opponent on a 3x3 board.
At the start of the game each player has three pieces occupying the row nearest to him. The pieces are similar to chess pawns, and may either be moved straight forwards into an empty square, or diagonally forwards to take an opponents piece.
A player loses if he cannot move any of his pieces.

Hexpawn is interesting to the computer programmer as it has the right level of complexity for experiments in computer 'learning' techniques. In the program given here, the program remembers each board position (or pattern) it encounters, and over the course of many games it learns the best move to make from each position. Given enough practice it can become very good.

When the computer has to make a move, it examines the array P() to see if it has encountered that position before, and if so then it chooses at random one of the moves held in P() against that position. If the position is new, then it updates P() with the new position and all possible legal moves it can make from that position before choosing one at random. If the computer loses a game, then it erases the move that led it to the losing position from P(), and similarly when it realises that there are no winning moves from a particular position, then it erases the move that led to that position. Thus, after enough games, P() will hold only winning moves.

The state of the board is held in the 9-element array B(); blank squares as 0, the computer's pieces as -1,and its opponent's as +1. The array M() is used to hold all legal moves at each position; each move being represented as a 2 digit number e.g. '25', the first digit being the 'From' location, the second being the location the piece is moved to. P() contains a list of entries, each representing a board position and the non-losing moves from it. The board position is encoded into a single positive number to save storage space (lines 4020-4040), the moves are stored as negative numbers, and the list is terminated by 0.

```
  1 REM HEXPAWN
  2 REM GO TO 200 TO KEEP SKILL

100 DIM P(100)
110 DIM B(8)
120 DIM M(4)
130 LET H=1
140 LET C=-1
200 REM START POS
210 FOR K=0 TO 2
```



```
220 LET B(K)=C
230 LET B(K+3)=0
240 LET B(K+6)=H
250 NEXT K
260 GO SUB 1000
270 LET PN=0
300 REM H MOVE
305 PRINT "YOUR MOVE: FROM TO"
310 INPUT I
320 LET F=I/10
330 LET T=I-F*10
340 LET TYP=H
350 GO SUB 2000
360 IF ERR>0 THEN GO TO 310
370 LET B(F)=0
380 LET B(T)=H
390 GO SUB 1000
400 REM C MOVE
410 LET TYP=C
420 GO SUB 3000
430 IF MOVE=0 THEN GO TO 600
440 PRINT "NEWLINE FOR MY MOVE"
450 INPUT I$
460 GO SUB 4000
470 LET B(F)=0
480 LET B(T)=C
490 GO SUB 1000
500 REM C WON ?
510 LET TYP=H
520 GO SUB 3000
530 IF MOVE>0 THEN GO TO 300
540 PRINT "I WIN"
550 GO TO 620
600 GO SUB 5000
610 PRINT "YOU HAVE WON"
620 PRINT
630 PRINT "NEWLINE FOR ANOTHER GAME"
640 INPUT I$
650 IF I$="" THEN GO TO 200
660 STOP
1000 REM DISPLAY
1010 CLS
1020 FOR K=0 TO 2
1030 LET X=3*K
1040 PRINT ,X;" ";X+1;" ";X+2,
1050 FOR Y=0 TO 2
1060 LET Z=B(X+Y)
1070 IF Z=0 THEN PRINT " ▦";
1080 IF Z=H THEN PRINT " H";
1090 IF Z=C THEN PRINT " C";
1100 NEXT Y
1110 PRINT
1120 PRINT
1130 NEXT K
1140 PRINT
1150 RETURN
2000 REM CHECK MOVE
2010 LET ERR=1
2020 IF F>8 OR T>8 THEN RETURN
```

```
2030 IF B(T)=TYP THEN RETURN
2040 IF NOT(B(F)=TYP) THEN RETURN
2050 IF NOT(F/3-T/3=TYP) THEN RETURN
2060 LET A=ABS(F-T)
2070 IF A=3 AND B(T)=0 THEN GO TO 2100
2080 IF (A=2 OR A=4) AND B(T)=-TYP THEN GO TO 2100
2090 RETURN
2100 LET ERR=0
2110 RETURN
3000 REM GET POSS MOVES
3010 LET MOVE=0
3020 FOR F=0 TO 8
3030 IF NOT(B(F)=TYP) THEN GO TO 3100
3040 FOR T=0 TO 8
3050 GOSUB 2000
3060 IF ERR>0 THEN GO TO 3090
3070 LET M(MOVE)=T+10*F
3080 LET MOVE=MOVE+1
3090 NEXT T
3100 NEXT F
3110 RETURN
4000 REM LOOK UP POS IN P()
4010 LET POS=0
4020 FOR K=0 TO 8
4030 LET POS=B(K)+1+3*POS
4040 NEXT K
4060 FOR K=0 TO 100
4070 LET PN=K
4080 IF P(PN)=0 THEN GO TO 4400
4090 IF P(PN)=POS THEN GO TO 4200
4095 NEXT K
4200 REM FOUND IT
4210 FOR K=1 TO 10
4220 IF P(PN+K)>-1 THEN GO TO 4240
4230 NEXT K
4240 IF K=1 THEN GO TO 4300
4250 LET PN=PN+RND(K-1)
4260 LET F=-P(PN)/10
4270 LET T=-P(PN)-10*F
4280 LET PL=PN
4290 RETURN
4300 REM LOSING POS
4310 GO SUB 5000
4320 LET PL=100
4330 LET MOVE=RND(MOVE)-1
4340 LET F=M(MOVE)/10
4350 LET T=M(MOVE)-10*F
4360 RETURN
4400 REM SAVE NEW POS + MOVES
4410 LET P(PN)=POS
4420 FOR K=1 TO MOVE
4430 LET P(PN+K)=-M(K-1)
4440 NEXT K
4450 LET PN=PN+RND(MOVE)
4460 GO TO 4260
5000 REM DEL BAD MOVE
5010 IF PL=100 THEN GO TO 5050
5020 FOR A=PL TO 99
5030 LET P(A)=P(A+1)
5040 NEXT A
5050 LET P(100)=0
5060 RETURN
```


Othello

OTHELLO is played between the computer (C) and its human opponent (H) on an 8x8 board. At the start of the game each player has two pieces in the centre area of the board.
H and C take turns to place a new piece in one of the vacant squares so as to 'capture' one or more opponents pieces. Opponents pieces are captured if they lie on a horizontal, vertical, or diagonal line terminated at one end by the piece just placed, and at the other by an existing piece belonging to the player making the move. Pieces captured change type to that of the player who has captured them.
Each move must capture at least one piece, or else the player must forfeit that move.

The game ends when all 64 squares have been filled, or when neither player can move. The winner is the player with the most pieces at the end of the game.

The computer displays the board after every move. The human player enters his move in terms of the row (1-8) and column (A-H) of the desired square e.g. 8A, or he may forfeit his move by entering nothing (just pressing the NEWLINE key). The program will not accept invalid moves. The result of the move is then displayed and the program waits for the player to press the NEW LINE key before making its own move and displaying the result of that. Entering "99" will end the game at any time.

```
  A B C D E F G H
1
2
3
4       C H
5       H C
6
7
8
```

Starting position

```
  A B C D E F G H
1
2
3         H
4       H H
5       H C
6
7
8
```

After player chose 3D

The algorithm used by the program to calculate the computer's next move is a very basic one, although it takes about 45 seconds to run, and allows the human player a good chance of winning.

Lines 100-190 are a subroutine which returns the value S of placing a new piece of type T (T= C or H) at point A(P). If F = 1 then the subroutine will also change the state of the captured pieces (lines 165-180).

Lines 200-295 form a second subroutine which displays the current state of the board and calculates the scores SC and SH.

These two subroutines are placed early in the listing to speed up the program.

Lines 300-380 calculate the optimum position for the computer to place its piece. If two positions are equally good, line 340 makes a random choice between them.

Lines 400-425 display the position after the computer has made its move.

Lines 500-595 get the player's move, check it for validity, then display the result. Lines 600-620 wait for him to realise what he has done before allowing the computer to proceed.

Lines 700-790 are only executed at the start of each game, to set up the starting positions. The state of the board is stored in the 100 element array A(), which is treated as a 10 x 10 array comprising an 8 x 8 board with a border. The array N() holds the 8 possible directions of movement from any square on the board.

```
  1 REM OTHELLO
  2 GO TO 700

100 LET S=0
105 IF NOT A(P)=0 THEN RETURN
110 FOR I=0 TO 7
115 LET Q=P
120 LET K=1
125 LET Z=N(I)
130 LET Q=Q+Z
135 IF A(Q)=0 THEN GO TO 185
140 IF A(Q)=T THEN GO TO 155
145 LET K=K+1
150 GO TO 130
155 LET S=S+K-1
160 IF F=0 THEN GO TO 185
165 FOR K=1 TO K
170 LET A(P+K*N(I))=T
175 NEXT K
180 LET A(P)=T
185 NEXT I
190 RETURN

200 LET SH=0
205 LET SC=0
210 PRINT
215 PRINT ,"  A B C D E F G H"
220 FOR R=1 TO 8
225 PRINT
230 PRINT ,R;
235 FOR V=1 TO 8
240 LET X=A(R+V*10)
```



```
245 IF X=C THEN PRINT " C";
250 IF X=H THEN PRINT " H";
255 IF X=0 THEN PRINT " ■";
260 IF X=C THEN LET SC=SC+1
265 IF X=H THEN LET SH=SH+1
270 NEXT V
275 PRINT
280 NEXT R
285 PRINT
290 PRINT "YOU HAVE ";SH;" I HAVE ";SC
295 RETURN

300 LET M=0
305 LET F=0
310 LET T=C
315 FOR R=1 TO 8
320 FOR V=1 TO 8
325 LET P=R+10*V
330 GOSUB 100
335 IF S<M THEN GO TO 355
340 IF S+RND(2)-1=M THEN GO TO 355
345 LET M=S
350 LET X=P
355 NEXT V
360 NEXT R
365 IF M=0 THEN GO TO 410
370 LET F=1
375 LET P=X
380 GOSUB 100


400 LET V=X/10
405 PRINT "I CHOSE ";X-10*V;CHR$(V+37)
410 IF M=0 THEN PRINT "I COULDNT MOVE"
415 GOSUB 200
420 IF H$="" AND M=0 THEN GO TO 1000
425 IF SC+SH=64 THEN GO TO 1000

500 PRINT "YOUR MOVE (E.G.8A)"
505 INPUT H$
510 IF H$="" THEN GO TO 585
515 IF H$="99" THEN GO TO 1000
520 LET R=CODE(H$)-28
525 IF R<1 OR R>8 THEN GO TO 505
530 LET A$=TL$(H$)
535 IF A$="" THEN GO TO 505
540 LET V=CODE(A$)-37
545 IF V<1 OR V>8 THEN GO TO 505
550 LET P=R+10*V
555 LET T=H
560 LET F=0
565 GO SUB 100
570 IF S=0 THEN GO TO 505
575 LET F=1
580 GO SUB 100
585 CLS
590 GO SUB 200
595 IF SC+SH=64 THEN GO TO 1000

600 PRINT "NEWLINE FOR MY MOVE"
605 INPUT A$
610 IF A$="99" THEN GO TO 1000
615 CLS
620 GO TO 300
```

```
700 DIM A(99)
705 DIM N(7)
710 LET N(0)=11
715 LET N(1)=10
720 LET N(2)=9
725 LET N(3)=1
730 LET N(4)=-1
735 LET N(5)=-9
740 LET N(6)=-10
745 LET N(7)=-11
750 LET C=-1
755 LET H=1
760 LET A(44)=C
765 LET A(55)=C
770 LET A(45)=H
775 LET A(54)=H
780 CLS
785 GO SUB 200
790 GO TO 500

1000 IF SC>SH THEN PRINT "I WON"
1005 IF SC<SH THEN PRINT "CONGRATULATIONS"
1010 IF SC=SH THEN PRINT "- A DRAW -"
```

Decimal Peeker

For those who have an overwhelming desire to examine the contents of the ZX80 memories (RAM or ROM), the Decimal Peeker will display the contents of 16 consecutive memory locations in decimal form (0-255). It also displays the decimal values of consecutive pairs of bytes, assuming them to represent a 16-bit signed integer having a decimal value in the range -32768 to +32767.

When the program has found and displayed the 16 memory locations, pressing NEWLINE will make it look at the next 16.

```
 10 REM DECIMAL PEEKER
 20 PRINT "ADDRESS (DECIMAL)"
 30 INPUT A
100 CLS
110 PRINT "ADDR   BYTE 1  BYTE 2  16 BIT"
120 PRINT
200 FOR L=1 TO 8
210 LET B1=PEEK(A)
220 LET B2=PEEK(A+1)
230 IF B2>127 THEN LET B3=-(B2-128)*256-B1
240 IF B2<128 THEN LET B3=B2*256+B1
250 PRINT A,B1,B2,B3
260 PRINT
270 LET A=A+2
280 NEXT L
300 PRINT
310 PRINT "NEWLINE FOR MORE"
320 INPUT A$
330 IF A$="" THEN GO TO 100
```


Hex Peeker

For those more used to working with Hexadecimal representations of memory contents, this program displays the values of 64 consecutive memory locations in Hex form.

The program cannot handle addresses above 7FFFh, which cause an arithmetic overflow in line 50, but this is not likely to worry the ZX80 user.

The display consists of 8 lines (counted by variable L), each containing the hex representations of 8 consecutive memory locations, preceeded by the address of the first byte in the line ; shown as 4 hex digits.

Hexadecimal representation splits each 8-bit byte into two 4-bit groups, and then represents each 4-bit group by a character in the range 0-9,A-F (0 = '0000', 9 = '1001', A = '1010', F = '1111').
Thus '11110000' (decimal 240) is represented as F0.
Programmers working in machine language prefer to use hexadecimal rather than decimal to represent the contents of memory locations as it more directly represents the states of the actual 'bits' in the byte.

```
  5 REM HEX PEEKER
 10 PRINT " ADDRESS (HEX)"
 20 INPUT A$
 30 LET A = 0
 40 FOR L = 1 TO 4
 50 LET A = CODE(A$) - 28 + 16 * A
 60 LET A$ = TL$(A$)
 70 NEXT L
100 CLS
110 FOR L = 1 TO 8
120 LET P = A/256
130 GO SUB 1010
140 LET P = A - P * 256
150 GO SUB 1010
160 PRINT "  ";
200 FOR A = A TO A + 7
210 LET P = PEEK(A)
220 GOSUB 1000
230 NEXT A
240 PRINT
250 PRINT
260 NEXT L
300 PRINT "NEWLINE FOR MORE"
310 INPUT A$
320 IF A$ = "" THEN GO TO 100
330 LIST
1000 PRINT " ";
1010 LET H = P/16
1020 PRINT CHR$(28+H);CHR$(28+P-H*16);
1030 RETURN
```

Animals

This program requires 3k bytes of RAM.

You think of an animal, the computer then tries to find out what it is by asking a series of questions to which you answer Y or N. If the computer doesn't know the animal you are thinking of it will finally ask you what it is, and also for a question to distinguish this animal from the others. These 'facts' are then stored so that the computer is better informed next time you play. To stop playing, enter N in response to the question 'ARE YOU THINKING OF AN ANIMAL ?'.

If you have loaded the program from tape, then starting it by GO TO 1 instead of RUN will preserve the data base it had when the program was saved.

Data is stored in 25 strings; A$ to Y$, which hold the questions and answers, and in a 25 element numeric array Q(24). Each element of Q holds information about the corresponding string (Q(0) for A$, Q(1) for B$ etc.). If Q() is zero, then the string contains the name of an animal. If the string contains a question, then Q() = 100*N + Y where N = the number (0-24) of the next string if the answer to the question is N; Y = the number of the next string if the answer is Y. See for example lines 110-160. F is the number of the first unused (free) string.

The program illustrates a straightforward - although rather long winded - way of overcoming the lack of string arrays in ZX80 4K BASIC. Lines 1000-1049 allow the string Z$ to be stored in one of 25 locations A$ to Y$ by doing a GO SUB 1000 + 2*P , where P is from 0 to 24. Similarly, lines 1050 to 1099 allow the program to retrieve the string by doing a GO SUB 1050 + 2*P.

Lines 110-170 make sure that the computer know about two animals to start with.

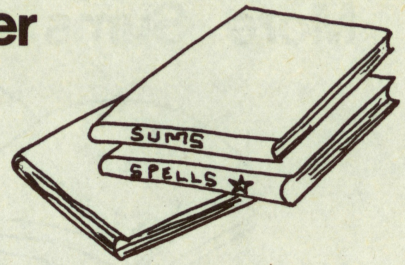
Lines 300-330 get one of the 25 strings. If it is a question then it is printed, otherwise the program jumps to line 400 to see if it has found the correct answer. If not, then lines 420-425 move the name of the animal the computer thought it was to the first free string (F), then lines 430-465 ask you for the name of your animal, store it in the next free string (F+2, see line 440), and ask for a question which is then stored (line 460).Lines 470-492 then determine a new value for Q(P) corresponding to the new question and animal you have told the computer about.

If lines 300-330 print a question - not the name of an animal - then lines 340-370 calculate which string to get next, depending on the answer to the question.

```
  1 REM ANIMALS
100 DIM Q(24)
110 LET A$="DOES IT SWIM"
120 LET B$="SPARROW"
130 LET C$="SHARK"
140 LET Q(0)=102
150 LET F=3


200 CLS
210 PRINT "ARE YOU THINKING OF AN ANIMAL ?"
220 LET P=0
230 INPUT Z$
240 IF NOT CODE(Z$)=CODE("Y") THEN STOP

300 CLS
310 GO SUB 1050+2*P
320 IF Q(P)=0 THEN GO TO 400
330 PRINT Z$;" ?"
340 INPUT Z$
350 LET N=Q(P)/100
360 LET P=Q(P)-100*N
370 IF CODE(Z$)=CODE("N") THEN LET P=N
380 GO TO 300
400 PRINT "IS IT A ";Z$;" ?"
405 INPUT Z$
410 IF CODE(Z$)=CODE("Y") THEN GO TO 200
415 IF F>24 THEN GO TO 500
417 CLS
420 GO SUB 1050+2*P
425 GO SUB 1000+2*F
430 PRINT "IT WAS A ?"
435 INPUT Z$
437 CLS
438 PRINT "WHAT QUESTION DISTINGUISHES"
439 PRINT "A ";Z$;
440 GO SUB 1002+2*F
445 GO SUB 1050+2*P
450 PRINT " FROM A ";Z$;" ?"
455 INPUT Z$
460 GO SUB 1000+2*P
465 GO SUB 1050+2*F
467 CLS
470 PRINT "FOR A ";Z$
472 PRINT "THE ANSWER WOULD BE ?"
475 INPUT Z$
480 LET Q(P)=101*F+1
485 IF CODE(Z$)=CODE("Y") THEN LET Q(P)=Q(P)+99
487 LET Q(F)=0
490 LET Q(F+1)=0
492 LET F=F+2
495 GO TO 200

500 PRINT "SORRY-MY MEMORY IS NOW FULL"
510 PRINT
520 GO TO 210
```

```
1000 LET A$=Z$
1001 RETURN
1002 LET B$=Z$
1003 RETURN
1004 LET C$=Z$
1005 RETURN
1006 LET D$=Z$
1007 RETURN
1008 LET E$=Z$
1009 RETURN
1010 LET F$=Z$
1011 RETURN
1012 LET G$=Z$
1013 RETURN
1014 LET H$=Z$
1015 RETURN
1016 LET I$=Z$
1017 RETURN
1018 LET J$=Z$
1019 RETURN
1020 LET K$=Z$
1021 RETURN
1022 LET L$=Z$
1023 RETURN
1024 LET M$=Z$
1025 RETURN
1026 LET N$=Z$
1027 RETURN
1028 LET O$=Z$
1029 RETURN
1030 LET P$=Z$
1031 RETURN
1032 LET Q$=Z$
1033 RETURN
1034 LET R$=Z$
1035 RETURN
1036 LET S$=Z$
1037 RETURN
1038 LET T$=Z$
1039 RETURN
1040 LET U$=Z$
1041 RETURN
1042 LET V$=Z$
1043 RETURN
1044 LET W$=Z$
1045 RETURN
1046 LET X$=Z$
1047 RETURN
1048 LET Y$=Z$
1049 RETURN
1050 LET Z$=A$
1051 RETURN
1052 LET Z$=B$
1053 RETURN
1054 LET Z$=C$
1055 RETURN
1056 LET Z$=D$
1057 RETURN
1058 LET Z$=E$
1059 RETURN
1060 LET Z$=F$
1061 RETURN
1062 LET Z$=G$
1063 RETURN
1064 LET Z$=H$
1065 RETURN
1066 LET Z$=I$
1067 RETURN
1068 LET Z$=J$
1069 RETURN
1070 LET Z$=K$
1071 RETURN
1072 LET Z$=L$
1073 RETURN
1074 LET Z$=M$
1075 RETURN
1076 LET Z$=N$
1077 RETURN
1078 LET Z$=O$
1079 RETURN
1080 LET Z$=P$
1081 RETURN
1082 LET Z$=Q$
1083 RETURN
1084 LET Z$=R$
1085 RETURN
1086 LET Z$=S$
1087 RETURN
1088 LET Z$=T$
1089 RETURN
1090 LET Z$=U$
1091 RETURN
1092 LET Z$=V$
1093 RETURN
1094 LET Z$=W$
1095 RETURN
1096 LET Z$=X$
1097 RETURN
1098 LET Z$=Y$
1099 RETURN
```


Sums Tester

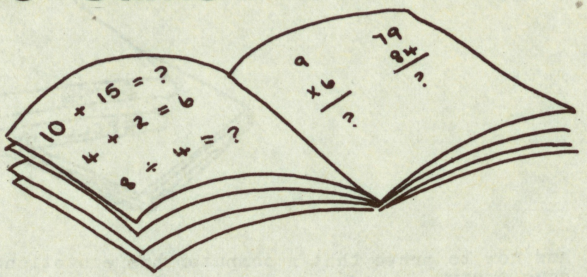


And now to prove that a computer has educational value; SUMS TESTER provides an exercise in simple addition. It poses 10 questions of the form A + B = ? , and displays a running total of the number of correct answers. At the end of the ten questions, it also shows the total time taken to answer them.

The program is based on the FOR - NEXT loop covering lines 130 to 490. Lines 300,310 set the ZX80 internal clock to zero when each question is displayed, and line 330 adds the time taken to answer to the variable TIME. Since the ZX80 internal clock is incremented 50 times per second, TIME is divided by 50 when printing out the total time taken in line 510.

```
  1 REM SUMS TESTER
100 RANDOMISE
110 LET SCORE=0
120 LET TIME=0
130 FOR G=1 TO 10
140 PRINT
150 PRINT "PRESS NEWLINE FOR QUESTION ";G
160 INPUT A$

200 CLS
210 LET A=RND(99)
220 LET B=RND(99)
230 LET ANSWER = A+B
240 PRINT ,A;" + ";B;" = ?"

300 POKE 16414,0
310 POKE 16415,0
320 INPUT ENTRY
330 LET TIME=TIME+PEEK(16414)+PEEK(16415)*256
400 PRINT
410 IF ENTRY=ANSWER THEN GO TO 450
420 PRINT ENTRY;" IS WRONG"
430 PRINT "THE CORRECT ANSWER IS ";ANSWER
440 GO TO 470
450 PRINT ENTRY;" IS CORRECT"
460 LET SCORE=SCORE+1
470 PRINT
480 PRINT "YOUR SCORE IS NOW ";SCORE;" OUT OF ";G
490 NEXT G
500 PRINT
510 PRINT "ALLTOGETHER YOU TOOK ";TIME/50;" SECONDS"
520 STOP
```

More Sums



The basic version of SUMS TESTER only poses questions of simple addition. The following modifications will make it cover simple subtraction, multiplication and division as well.

Line 210 now chooses at random whether the problem is to be one of addition, subtraction, multiplication or division, and calls one of the subroutines (1000- , 2000- , 3000- or 4000-) accordingly. Whichever subroutine is called does the work of determining and displaying the problem, and also calculates the correct answer.

The multiplication and division problems assume a knowledge of the 12 times table, and won't ask trivial questions such as 3 * 1 = ?

Change line 210 to;

```
210 GO SUB RND(4)*1000
```

Delete lines 220,230,240

Add the following;

```
1000 LET A=RND(99)
1010 LET B=RND(99)
1020 LET ANSWER=A+B
1030 PRINT ,A;" + ";B;" = ?"
1040 RETURN
2000 LET A=RND(99)
2010 LET B=RND(90)
2020 IF B+2>A THEN GO TO 2000
2030 LET ANSWER=A-B
2040 PRINT ,A;" - ";B;" = ?"
2050 RETURN
3000 LET A=RND(11)+1
3010 LET B=RND(11)+1
3020 LET ANSWER=A*B
3030 PRINT,A;" TIMES ";B;" = ?"
3040 RETURN
4000 LET A=RND(12)
4010 LET ANSWER=RND(11)+1
4020 PRINT ,ANSWER*A;" / ";A;" = ?"
4030 RETURN
```


Weekday

This program calculates the day of the week for any date after 1750 AD (except for Russia, which did not change to the Gregorian calendar until 1917).

Monday's child is fair of face,
Tuesday's child is full of grace.
Wednesday's child is full of woe,
Thursday's child has far to go.
Friday's child is loving and giving,
Saturday's child works hard for a living,
But the child that's born on the Sabbath day
Is bonny, blithe, and good, and gay.

```
 1 PRINT "DAY OF WEEK"
10 DIM K(12)
11 LET K(1)=0
12 LET K(2)=3
13 LET K(3)=3
14 LET K(4)=6
15 LET K(5)=1
16 LET K(6)=4
17 LET K(7)=6
18 LET K(8)=2
19 LET K(9)=5
20 LET K(10)=0
21 LET K(11)=3
22 LET K(12)=5

30 PRINT
32 PRINT "YEAR (>1750) ";
34 INPUT Y
36 IF Y<1750 THEN GO TO 34
38 PRINT Y
40 PRINT "MONTH (1-12) ";
42 INPUT M
44 IF M<1 OR M>12 THEN GO TO 42
46 PRINT M
50 PRINT "DAY   (1-31) ";
52 INPUT D
54 IF D<1 OR D>31 THEN GO TO 52
56 PRINT D
60 LET Y1=Y-100*(Y/100)
61 LET X=Y/100-15
62 LET A=3+X*21
64 LET I=(A-28*(A/28))/4
66 LET A=Y1*5+(D+K(M)+I)*4
68 LET W=(A-28*(A/28))/4+1
70 IF M>2 THEN GO TO 82
72 LET L=X-1+4*((X-1)/4)
74 IF Y1>0 THEN LET L=Y-4*(Y/4)
76 IF L>0 THEN GO TO 82
78 IF W=0 THEN LET W=6
80 LET W=W-1
82 IF W=0 THEN LET W=7

90 PRINT
92 PRINT
94 PRINT "IS A ";
96 GO SUB 98+2*W
98 STOP

100 PRINT "SUNDAY"
101 RETURN
102 PRINT "MONDAY"
103 RETURN
104 PRINT "TUESDAY"
105 RETURN
106 PRINT "WEDNESDAY"
107 RETURN
108 PRINT "THURSDAY"
109 RETURN

110 PRINT "FRIDAY"
111 RETURN
112 PRINT "SATURDAY"
113 RETURN
```

Music

This program lets you store a short tune in RAM and play it out to a tape recorder or amplifier connected to the ZX80's 'MIC' jack socket.

The program runs on a machine with 2k bytes of RAM, and can store a sequence of up to 233 'parts', each 'part' being either a single note or a quiet period between notes.

The parts may be entered or updated in any order, but the program plays out the tune by starting with part 1, then continuing with parts 2,3 etc until the end of the tune is reached.

For each part you must enter a Time; represented by any number in the range 1 to 255, which defines how long that note or pause will last. A 'Time' of 0 signifies the end of the tune.

A 'Note' number has also to be entered for each part. The value 0 gives a silent period, while values from 1 to 255 give differing frequencies; the higher values giving the lower notes.
To use the program, first work out the 'parts', and their associated Time and Note values; as in the examples. Then Run the program to enter the values.

For each part, the program will display;

```
PART (1-233, 0=PLAY)  n
```

where n is 1 when you first run the program, and thereafter is one greater than the number of the part previously entered. If n is the number of the part you wish to enter or change, just press NEWLINE, otherwise enter the required part number. The display will then show;

```
TIME (1-255, 0=END)    x
```

where x is the value that was previously stored for this part. Enter the desired Time value, or just press NEWLINE if you don't want to change the old value. The program will then display;


```
NOTE (1-255, 0=QUIET)  y
```

where y is the value previously stored. This may be changed or left with the same value by either entering the new value or just NEWLINE.

When all parts of the tune have been entered, then it can be played by entering a 'part number' of 0. Quiet periods, each lasting for a few seconds, are generated before and after playing the tune (by lines 81,82 & 84,85) to separate the tune from the TV synchronisation noise normally produced by the ZX80.

When the tune has been played, then the message

```
ENTER 1 TO QUIT, 0 TO KEEP TUNE
```

will appear. If the tune is kept, it may be amended by just entering the new values for the parts to be changed. If the program is saved on tape after storing a tune, then the tune will be saved along with the program and on re-loading the program, the tune may be kept by starting the program with GO TO 5, rather than RUN.

```
 1 REM MUSIC
 2 DIM A(255)
 5 LET A=PEEK(16392)+PEEK(16393)*256+2
10 LET A(0)=11009
11 LET A(1)=2304
12 LET A(2)=62
13 LET A(3)=17955
14 LET A(4)=-14152
15 LET A(5)=22051
16 LET A(6)=10426
17 LET A(7)=7190
18 LET A(8)=17355
19 LET A(9)=1064
20 LET A(10)=211
21 LET A(11)=1048
22 LET A(12)=219
23 LET A(13)=62
24 LET A(14)=-18421
25 LET A(15)=-6616
26 LET A(16)=8213
27 LET A(17)=6393
28 LET A(18)=3046
29 LET A(19)=10424
30 LET A(20)=221
31 LET A(21)=-8168
35 LET P=1
40 CLS
41 PRINT "PART (1-233, 0=PLAY) ";P
42 GO SUB 100
43 IF X<0 THEN GO TO 50
44 IF X=0 THEN GO TO 80
45 IF X>233 THEN GO TO 42
46 LET P=X
47 CLS
48 PRINT "PART (1-233, 0=PLAY) ";P
50 LET X=PEEK(A+42+2*P)
51 PRINT "TIME (1-255, 0=END)   ";X;" ";
52 GO SUB 100
```

```
53 IF X<0 THEN PRINT
54 IF X<0 THEN GO TO 60
55 IF X>255 THEN GO TO 52
56 PRINT X
57 POKE A+42+2*P,X
58 IF X=0 THEN GO TO 70
60 LET X=PEEK(A+43+2*P)
61 PRINT "NOTE (1-255, 0=QUIET) ";X
62 GO SUB 100
63 IF X<0 THEN GO TO 70
64 IF X>255 THEN GO TO 62
65 POKE A+43+2*P,X
70 LET P=P+1
71 GO TO 40
80 CLS
81 FOR X=1 TO 500
82 NEXT X
83 LET X=USR(A)
84 FOR X=1 TO 500
85 NEXT X
86 PRINT "ENTER 1 TO QUIT, 0 TO KEEP TUNE"
87 INPUT X
88 IF X=0 THEN GO TO 35
89 STOP

100 LET X=0
101 INPUT X$
102 IF X$="" THEN LET X=-1
103 IF X$="" THEN RETURN
104 LET X=X*10+CODE(X$)-28
105 LET X$=TL$(X$)
106 GO TO 103
```

Notes

The system of tuning known as 'equal temperament' has been used on keyboard and fretted instruments since the 16th century. In this system an octave is divided into twelve equal intervals or 'semitones'. The ratio between the frequencies of two notes one semitone apart is therefore equal to the twelfth root of 2;

$$\frac{f_{n+1}}{f_n} = \sqrt[12]{2} \simeq 1.0594631$$

Using the division ratios from 1 to 255 provided by MUSIC, a set of rules can be derived which are good approximations to the familiar equal tempered scale. Table 1 shows just such a sequence covering two octaves, based on the lowest note obtainable, which is approximately F#.

The second column of Table 1 shows the 'Note' number to be entered; the third column shows the equal tempered interval required and the fourth column shows the actual interval obtained. It will be noted that the accuracy of the approximation gets worse at higher frequencies. Transposition can easily be achieved by offseting the first column by the desired interval.

	'NOTE'	Required	Achieved
F#	255	1.0000	1.0000
G	241	1.0594	1.0581
G#	227	1.1224	1.1233
A	214	1.1892	1.1915
A#	203	1.2599	1.2561
B	191	1.3348	1.3351
C	180	1.4142	1.4166
C#	170	1.4983	1.5000
D	161	1.5874	1.5838
D#	151	1.6818	1.6887
E	143	1.7818	1.7832
F	135	1.8877	1.8884
F#	127	2.0000	2.0078
G	120	2.1189	2.1250
G#	113	2.2449	2.2566
A	107	2.3784	2.3831
A#	101	2.5198	2.5247
B	96	2.6696	2.6562
C	90	2.8284	2.8333
C#	85	2.9966	3.0000
D	80	3.1748	3.1875
D#	76	3.3636	3.3552
E	72	3.5636	3.5417
F	68	3.7754	3.7500
F#	64	4.0000	3.9804

Tempo

The rate of playing a musical piece; its Tempo, is usually given in terms of Maelzel's metronome markings; the number of notes of a given measure to be played in 1 minute. Thus ♩ = 90 represents 90 crochets per minute, which is a slow medium tempo. The time values to be inserted in your program for this tempo would be 240 for a crochet (♩), 120 for a quaver (♪) and so on (the value 255 gives a duration of about 0.7 seconds). To achieve long notes, two or more parts can be consecutively entered with the same value of pitch. If you wish to sound separate, consecutive, notes of the same pitch you must insert a short pause at the end of each note to distinguish one from the other. The total time for note plus pause must equal the nominal value for the note to be played. Longer pauses of this type may be associated with any note and give rise to a staccato effect. For faster tempos decrease the above figures, keeping the same ratios.

Machine code

The notes are generated by a machine language routine which takes the Note and Time values loaded by the BASIC program into the array A() space. Lines 10 to 31 of MUSIC actually load this machine code into RAM - as described in the 'Using USR' section of this book - then line 83 runs it.

The machine language routine used is listed on the following page. It generates the tone by flipping the ZX80's SYNCH line at a rate determined by the 'Note' value stored in RAM.

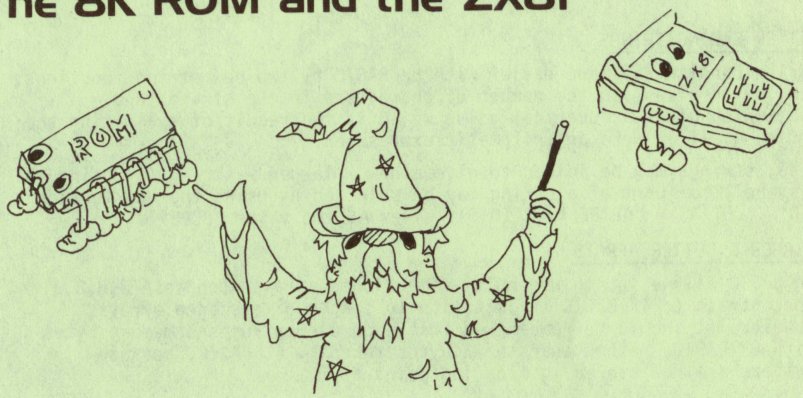
```
01 2B 00        START   LD BC 43        POINT HL TO START
09                      ADD HL  BC      OF DATA
3E 00                   LDA  0
23              NUNOTE  INC HL
46                      LD B  (HL)      GET NEXT TIME
B8                      CP  B
C8                      RET Z           QUIT IF ZERO
23                      INC HL
56              CYCLE   LD D  (HL)
BA                      CP  D           GET NOTE VALUE
28 16                   JRZ  QUIET      QUIET IF ZERO
1C                      INC E
CB 43                   BIT 0  E
28 04                   JRZ TON
D3 00                   OUT 0  A        ONE HALF CYCLE
18 04                   JR TIME
DB 00           TON     IN 0  A         OTHER HALF CYCLE
3E 00                   LDA  0
0B              TIME    DEC BC          END OF NOTE ?
B8                      CP  B
28 E6                   JRZ NUNOTE
15                      DEC D           END OF HALF CYCLE ?
20 F9                   JRNZ TIME
18 E6                   JR CYCLE
0B              QUIET   DEC BC
B8                      CP B            END OF QUIET PERIOD ?
28 DD                   JRZ NUNOTE
00                      NOP             TIME WASTER
18 E0                   JR CYCLE
```

Examples

PART	TIME	NOTE
1	240	90
2	240	107
3	240	135
4	240	180
5	80	161
6	80	143
7	80	135
8	160	161
9	80	135
10	240	180
11	220	180
12	20	0
13	240	120
14	240	90
15	240	107
16	240	135
17	80	161
18	80	143
19	80	135
20	160	120
21	80	107
22	160	120
23	160	120
24	0	

PART	TIME	NOTE
1	40	214
2	80	241
3	80	214
4	80	180
5	80	214
6	80	241
7	80	214
8	120	180
9	40	214
10	80	161
11	80	143
12	80	161
13	80	143
14	240	161
15	80	143
16	80	161
17	80	143
18	80	120
19	80	143
20	80	161
21	80	143
22	120	120
23	40	143
24	80	180
25	80	161
26	80	180
27	80	214
28	240	180
29	0	

The 8K ROM and the ZX81



What's New ?

If you have bought a ZX81, or have already fitted the new 8K ROM to your ZX80, then you will already know all about Sinclair's 8K BASIC. However, for those readers who bought the original ZX80 with the 4K ROM, and are wondering whether to upgrade it - and also because the programs in this book were originally written for the 4K ROM version of BASIC, a summary of the main differences between the two BASICs is called for.

THE KEYBOARD

Both have the same number of keys, but the 8K BASIC provides more functions on each key. Anyone buying the 8K ROM to upgrade his ZX80 should also receive a new keyboard overlay.

INTEGER vs FLOATING POINT

4K BASIC does all it's arithmetic using only Integers (whole numbers) between -32768 and +32767. The 8K BASIC, on the other hand, uses decimal numbers (such as 12.345), calculates to an accuracy of about 9 decimal digits, and will accept numbers as large as about 10^{38} - useful in these days of rampant inflation.

Some programs (examples in this book include Matches and Weekday) rely on the properties of Integer arithmetic - particularly Integer division, and to accomodate them 8K BASIC has the function INT which converts a floating point number to it's Integer equivalent.

MORE MATHS FUNCTIONS

8K BASIC provides the following additional functions;

SQR - square root
SIN - sine
COS - cosine
ASN - arcsine
ACS - arccosine
TAN - tangent
ATN - arctangent
LN - log to base e
EXP - exponential function
PI - gives the value 3.141592654
SGN - gives -1, 0 or +1 depending on the sign of it's argument

RND on 8K BASIC gives a decimal fraction between 0 and 0.99999...

BETTER STRING THINGS

String handling is made easier with 8K BASIC by two new string functions; LEN, which gives you the number of characters in the string, and VAL, which gives a numerical value which is the result of evaluating the string as if it were an arithmetic expression.

Also, strings may be joined to give a new - larger - string (by using the + symbol), or part of a string may be selected by using the new funtion 'TO' (LET A$ = "ABCDEF" (2 TO 5) gives A$ the value "BCDE").

BIGGER & BETTER ARRAYS

8K BASIC allows you to use multi-dimensioned arrays such as A(3,3,3,3), in contrast to 4K BASIC's limitation to single dimensioned arrays. However, it should be remembered that each element uses 5 memory bytes with 8K BASIC, rather than the 2 bytes taken by 4K BASIC, because numbers are now stored in floating point form.

String arrays are also allowed by 8K BASIC, and again they may be multi dimensioned.

NAMES FOR PROGRAMS

8K BASIC allows you to give a name to a program when you save it on tape. Then, by including that name in a LOAD command, the tape will be searched for the program with that name.

BETTER PRINTING & PLOTTING

Two new print functions make formatting of the display much easier;

AT line,col ;- starts printing the next item at the specified line and column.

TAB col ;- starts printing the next item at the specified column.

The new SCROLL command moves the whole display up one line (losing the previous top line).

Plotting graphs - by using the black and white graphics characters to give a 64 x 40 grid - is made easier by the two new commands;

PLOT x,y ;- plot a black square at point x,y

UNPLOT x,y ;- plot a white square at point x,y

When the ZX printer arrives, then 8K BASIC's LPRINT,LLIST and COPY commands will let you drive it.

BETTER INTERACTION

The PAUSE n statement provided by 8K BASIC displays the picture (by stopping execution of your program) for n/50 seconds or until a key is pressed. At the end of this time the program resumes.

INKEY$ looks at the keyboard and returns a single character if a key was being pressed, or the null string if no key was pressed.

FAST & SLOW

The ZX81 can be operated at two speeds. The FAST mode is like the ZX80's way of operating, in which the display blanks out while your program is running, and flickers whenever you press a key. In the SLOW mode, priority is given to maintaining the TV display, and your program runs only during the spare time left at the end of each TV frame.

Unfortunately, a ZX80 fitted with the new 8K ROM will not operate in the SLOW mode.

Translation for 8K

To run the programs given in this book on the ZX81, or on a ZX80 fitted with the new 8K ROM, enter the program lines shown in the following paragraphs in place of the original versions.

RUSSIAN ROULETTE

```
300 IF RND<.17 THEN GOTO 400
420 PRINT ,"▄▄▄▄"
430 PRINT ,"▌▌▌▌"
440 PRINT ,"▌▌▌█"
450 PRINT ,"▀▀▀▀"
```

MATCHES

```
100 LET M = INT(21+12*RND)
400 LET R = M-4*INT(M/4)
410 IF R = 1 THEN LET C = 1 + INT(3 * RND)
420 IF R<>1 THEN LET C = R + 3 - 4 * INT((R + 3)/4)
```

NUMBER GUESSING A

```
30 LET N = INT(9 * RND + 1)
```

NUMBER GUESSING B

```
30 LET N = INT(99 * RND + 1)
```

DICE(S)

```
10 LET N = INT(6 * RND + 1)
1000 IF L>1 AND L<9 THEN PRINT "▐";
1045 IF N = 2 * (INT(N / 2)) + 1 THEN GOTO 1085
1065 PRINT "▀▀▀▀▀▀▀▀";
(delete line 1066)
```

For two die-

```
10 LET N1 = INT(6 * RND + 1)
15 LET N2 = INT(6 * RND + 1)
```

HORSE RACE

```
620 LET N1 = N1 + INT(5 * RND)
630 LET N2 = N2 + INT(5 * RND)
1040 IF R = 2 THEN PRINT "▄▄";
1050 IF R = 3 THEN PRINT "▀▀";
1060 IF R = 4 THEN PRINT "▌▌";
```

BUZZ-WORD

```
1000 LET N = 1 + INT(4 * RND)
1030 LET A$ = A$(2 TO )
1050 LET A$ = A$(2 TO )
1120 LET A$ = A$(2 TO )
```

MOON LANDER

```
240 PRINT INT(H / 10);TAB 8;INT(V / 10);TAB 16;INT F;
270 PRINT TAB 24;INT R
550 PRINT "AT ";INT(V / 10);" FEET/SEC"
560 PRINT "WITH ";INT F;" FUEL LEFT"
```

HIGHER/LOWER

```
120 LET N1 = 1 + INT(99 * RND)
130 LET N2 = 1 + INT(99 * RND)
```

RND

```
120 LET N = INT(1 + 20 * RND)
```

MEASLES

```
240 LET B = INT(1 + 128 * RND)
```

HAMMURABI

```
300 LET L = 17 + INT(8 * RND)
700 LET R = INT(S / 2 * RND)
710 LET C = INT(4 * P * RND)
730 LET D = INT(H - F / 15)
750 LET N = INT(H / 10 + H / 20 * RND)
780 IF RND<.17 THEN LET T = INT(H / 8 + H / 3 * RND)
```

NIBBLER

```
320 LET B = 1 + INT(30 * RND)
```

HEXPAWN

```
3 FAST
110 DIM B(9)
120 DIM M(5)
210 FOR K = 1 TO 3
270 LET PN = 1
320 LET F = INT(I/10)
1030 LET X = 3 * K + 1
1040 PRINT TAB 8;X;" ";X + 1;" ";X + 2;TAB 16;
2020 IF F>9 OR T>9 OR F<1 OR T<1 THEN RETURN
2050 IF (INT((F - 1)/3) - INT((T - 1)/3))<>TYP THEN RETURN
3020 FOR F = 1 TO 9
3040 FOR T = 1 TO 9
3070 LET M(MOVE + 1) = T + 10 * F
4020 FOR K = 1 TO 9
4060 FOR K = 1 TO 100
4250 LET PN = PN + INT((K - 1) * RND) + 1
4260 LET F = INT(-P(PN)/10)
4330 LET MOVE = INT(MOVE * RND)
4340 LET F = INT(M(MOVE + 1)/10)
4350 LET T = M(MOVE + 1) - 10 * F
4430 LET P(PN + K) = - M(K)
4450 LET PN = PN + INT(1 + MOVE * RND)
```

OTHELLO

```
2 FAST
3 GOTO 700
110 FOR I = 1 TO 8
215 PRINT TAB 10;"A B C D E F G H"
230 PRINT TAB 8;R;
240 LET X = A(1 + R + V * 10)
325 LET P = R + 10 * V + 1
340 IF S + INT(2 * RND) = M THEN GOTO 355
400 LET X = X - 1
402 LET V = INT(X/10)
530 LET A$ = H$( 2 TO )
550 LET P = R + 10 * V + 1
700 DIM A(100)
705 DIM N(8)
710 LET N(8) = 11
760 LET A(56) = H
775 LET A(46) = C
```

DECIMAL PEEKER

```
250 PRINT A;TAB 8;B1;TAB 16;B2;TAB 24;B3
```

HEX PEEKER

```
60 LET A$ = A$(2 TO )
120 LET P = INT(A / 256)
1010 LET H = INT(P / 16)
```


SUMS TESTER

```
210 LET A = 1 + INT(99 * RND)
220 LET B = 1 + INT(99 * RND)
240 PRINT A;" + ";B;" = ?"
300 POKE 16437,255
310 POKE 16436,255
330 LET TIME = TIME + 65535 - PEEK 16436 - 256 * PEEK 16437
```

MORE SUMS

```
210 GOSUB 1000 * (1 + INT(4 * RND))
1000 LET A = 1 + INT(99 * RND)
1010 LET B = 1 + INT(99 * RND)
2000 LET A = 1 + INT(99 * RND)
2010 LET B = 1 + INT(90 * RND)
3000 LET A = 2 + INT(11 * RND)
3010 LET B = 2 + INT(11 * RND)
4000 LET A = 1 + INT(12 * RND)
4010 LET ANSWER = 2 + INT(11 * RND)
```

MUSIC

Replace lines 1-31 inclusive by;

```
1 REM MUSIC
2 DIM A(104)
5 FAST
10 LET C$="2A1040013400093E002346B8C82356BA2816
1CCB432804D3FF1804DBFE3E000BB828E61520F918E60BB
828DD0018DF"
12 LET A = PEEK(16400) + 256 * PEEK(16401) + 6
14 FOR I = 0 TO 46
16 LET B = CODE(C$(2 * I + 1 TO )) - 28
18 LET B = CODE(C$(2 * I + 2 TO )) - 28
20 POKE A + I,C + 16 * B
22 NEXT I
24 LET A = A + 3
```

Change lines 83, 105 to;

```
83 LET X = USR(A - 3)
105 LET X$ = X$(2 TO )
```

WEEKDAY

```
60 LET Y1 = Y - 100 * INT(Y / 100)
61 LET X = INT(Y / 100) - 15
64 LET I = INT((A - 28 * INT(A / 28)) / 4)
68 LET W = INT((A - 28 * INT(A / 28)) / 4) + 1
72 LET L = X - 1 + 4 * INT((X - 1) / 4)
74 IF Y1>0 THEN LET L = Y - 4 * INT(Y / 4)
```

PLOTTING (page 34)

```
7130 POKE PEEK(16396) + 256 * PEEK(16397) + 1 + 33 *
     (19 - INT X) + Y,XP
7440 LET X$ = X$(2 TO )
```

MICROMON

```
2 DIM A(100)
16 LET A$ = A$(2 TO )
21 LET P = INT(A / 256)
35 POKE A - 1,16 * L + CODE(A$(2 TO )) - 476
50 LET L = INT(P / 16)
```

Animals

The 'Animals' program given in pages 22-24 of the book went to great lengths to overcome the lack of string arrays in 4K BASIC, the version listed below has been completely re-written to take advantage of 8K BASIC's features.

As the original version, this program tries to guess the name of an animal by asking a series of questions. Over a series of games it gradually builds up a database of animal names and suitable questions. When the program is loaded from tape, then starting it by GOTO 200 instead of RUN will preserve the database available when you saved it.

```
  1 REM ANIMALS
100 DIM A(50,2)
110 DIM A$(50,32)
120 LET A$(1)="DOES IT SWIM"
130 LET A$(2)="SHARK"
140 LET A$(3)="SPARROW"
150 LET A(1,1)=2
160 LET A(1,2)=3
170 LET B=4
200 LET P=1
210 CLS
220 PRINT "THINK OF AN ANIMAL"
230 PRINT
240 GOTO 310
300 CLS
310 IF A(P,1)=0 THEN GOTO 400
320 PRINT A$(P)
330 INPUT B$
340 LET B$=B$( TO 1)
350 IF B$<>"Y" AND B$<>"N" THEN GOTO 330
360 LET D=P
370 LET P=A(P,1)
380 IF B$="N" THEN LET P=A(D,2)
390 GOTO 300
400 PRINT "IS IT A"
410 PRINT A$(P)
420 INPUT C$
430 LET C$=C$( TO 1)
440 IF C$="Y" THEN GOTO 900
450 IF C$ <>"N" THEN GOTO 420
500 CLS
510 PRINT "OK - I GIVE UP"
520 PRINT "IT WAS A ?"
530 INPUT D$
540 IF LEN D$>32 THEN LET D$=D$( TO 32)
600 CLS
610 PRINT "WHAT QUESTION DISTINGUISHES A"
620 PRINT D$
630 PRINT "FROM A"
640 PRINT A$(P)
650 INPUT E$
660 IF LEN E$ >32 THEN LET E$=E$( TO 32)
700 PRINT
710 PRINT "FOR A"
720 PRINT D$
730 PRINT "WHAT IS THE ANSWER TO"
740 PRINT E$
```



```
750 INPUT C$
760 LET C$=C$( TO 1)
770 IF C$<>"Y" AND C$<>"N" THEN GOTO 750
800 LET A$(B)=E$
810 LET A$(B+1)=D$
820 IF B$="Y" THEN LET A(D,1)=B
830 IF B$="N" THEN LET A(D,2)=B
840 LET A(B,1)=B+1
850 LET A(B,2)=P
860 IF C$="N" THEN LET A(B,1)=P
870 IF C$="N" THEN LET A(B,2)=B+1
880 LET B=B+2
890 GOTO 940
900 CLS
910 PRINT "I GUESSED RIGHT"
920 PRINT
930 PRINT "- SUCH A CLEVER COMPUTER -"
940 IF B>50 THEN PRINT "MY MEMORY IS FULL"
950 PRINT
960 PRINT "ANOTHER GAME ?"
970 INPUT C$
980 IF C$( TO 1)="Y" THEN GOTO 200
990 STOP
```

Other 4K programs

Since 8K BASIC for the ZX80/81 is an enhanced version of the original 4K BASIC, programs written for the latter should need only a few changes to run with the new ROM - the only real problem which could arise is lack of RAM space because the more powerful BASIC takes up more memory for the system variables, uses 5 bytes instead of 2 for each numeric variable, and needs an extra 2 bytes to store each line of your program.

Given enough RAM, then most 4K BASIC programs can be converted by following the following rules;

- Put INT before each division.
  E.g. LET A=B + C/D becomes LET A=B + INT(C/D)
- Replace TL$(X$) by X$(2 TO).
  E.g. LET A$=TL$(B$) becomes LET A$=B$(2 TO)
- Replace commas in PRINT statements by the appropriate TAB statements.
  E.g. PRINT A,B,C becomes PRINT A;TAB 8;B;TAB 16;C
- Recalculate the value in each CHR$ command - as the values for some of the graphics symbols have been changed. In practice there should be less need for CHR$ as the new ROM allows you to enter all of the graphics symbols from the keyboard.
- Modify RND statements.
  E.g. LET A=RND(10) becomes LET A=1+INT(10*RND)
- Change the addresses used by PEEKs and POKEs to the new locations of the system variables.

Of course, these rules don't really take advantage of the more powerful features of the 8K BASIC, and it is ofet worth seeing if a simpler (and less memory consuming !) program can be obtained by spending some time on a more detailed study of how the original program works, and how it could be implemented in the better BASIC. For an example of how this approach can cut the length of a program, see the 4K BASIC version of 'Animals' given in the book, and also the 8K BASIC version in this supplement.

Other BASICs

Readers wanting to translate programs written in 'alien' BASICs will find that most of the difficulties discussed in pages 42-47 of the book are overcome by the new ROM. Thus we are now able to use multi-dimensioned numeric and string arrays, can handle floating point numbers, and have the more powerful arithmetic and trigonometrical functions.

The difficulties that remain are mainly concerned with the lack of DEF, FN,READ,DATA and RESTORE commands - and in these areas the hints given in the body of the book are still valid. We still do not have the string functions LEFT$,MID$ and RIGHT$ - which operate on a string to give a new string which is a part of the original - but they can easily be replaced by variations of the string TO command;

LEFT$(X$,Y) which gives the first Y characters from X$, translates to X$(TO Y).

RIGHT$(X$,Y) gives the last Y characters in X$, and translates to X$((LEN(X$)-Y) TO)

MID$(X$,Y,Z) gives Z characters from X$, starting at position Y. It's ZX81 equivalent is X$(Y TO (Y+Z))

When using ASC or CHR$, note that the ZX80/81 character codes still differ from the more standard ASCII codes used by most computers.

RAM

The 4K and 16K RAM extension circuits given on pages 60-64 will also work with the ZX81 - although the 1K resistor used in the 4K RAM extension should really be reduced to 470 ohms. Sinclair's 16K RAM pack will also work with the ZX81, although the case will have to be removed from early models of the RAM pack to allow it to fit properly to the rear of the ZX81.

Although the ZX81 manual says that the old 1-3K static RAM pack will not work with the ZX81, experiment has shown that it is useable if pin 23B of the connector (that's the pin at the bottom left-hand corner when viewing the RAM pack PCB from the component side) is removed, or if the track leading to it is cut. This modification does not affect it's working with the ZX80.

Other bits of the book

The notes 'Breaking Free' and 'Beware of Lone REMs' on pages 40-41 referred to problems with 4K BASIC, and do not apply to machines with the new 8K ROM.

Similarly, the hardware notes on pages 56-58 do not apply to the ZX81.

For 8K BASIC users, the table at the top of page 53 becomes;

location	value	
16509	0	
16510	1	Line number = 1
16511	6	
16512	0	6 bytes in line
16513	254	REM
16514	38	A
16515	39	B
16516	40	C
16517	41	D
16518	118	End of line

Plotting

There are three basic approaches to plotting a graph or drawing a pattern on the screen of a ZX80 system;

- Using a host of PRINT statements tailored to produce the required pattern. This is fine for small or regular patterns, but doesn't give you much flexibility.
- Storing an 'image' of the screen in an Integer Array, and Printing it when complete. This method gives great flexibility; especially for graphical applications, and allows you to change any part of the display easily. However, it does use a lot of memory. To store the image of an area of screen 20 lines by 32 columns takes a 640 element array, which uses over 1K bytes of RAM.
- Writing the pattern directly into the RAM area used for the screen display refresh. This method gives you all the flexibility of the second method, without the overhead of a large array.

We can write into the display area easily enough by using the POKE command, and read back if required with a PEEK, provided that we know the address of the memory byte corresponding to a particular position on the screen. This is not quite straightforward as the area of RAM used for the display refresh doesn't start at a fixed address, but is moved around by the ZX80 to make best use of the available RAM. Also, to conserve memory, the ZX80 doesn't reserve 32 memory locations for each line, but only enough to hold the characters actually printed to that line. Within the display area of RAM, each line on the screen can be from 0 to 32 memory bytes, followed by and 'end of line' byte; code 118. However, these difficulties are easily overcome by;

- Noting that if we add 1 to the number stored in the 'System Variable' locations 16396 and 16397, this gives us the address of the first character in the display area, corresponding to the top left hand corner of the screen.
- Reserving as much space as is needed in the display area of RAM by Printing complete 32 character lines of spaces.

Some general purpose subroutines embodying these principles are given on the next page.

The subroutine starting at line 7000 should be called first as it clears the screen, then writes 20 completely blank lines to reserve a 20 x 32 character area. Note that there should be 8 spaces between the " signs in line 7030. XP is later used as the code of the character to be printed; line 7050 gives it a default value of 128 (a black square).

The second subroutine starts at line 7100 and displays a single character (code XP) at a point defined by the values of X (0-19) and Y (0-31), overwriting any previous character in that place. 0,0 is the bottom left hand corner of the 20 x 32 element display area.

Lines 7200-7370 are a third subroutine which draws a straight line at any angle, starting at X0,Y0 and finishing at X1,Y1.

The final subroutine starts at line 7400 and displays the character string X$ starting at point X,Y. By changing the routine slightly you can make it write vertically, diagonally, or backwards.

The display lines below the reserved area are still useable by normal PRINT or for INPUT.

```
7000 REM INITIALISE
7010 CLS
7020 FOR X=1 TO 80
7030 PRINT "        ";
7040 NEXT X
7050 LET XP=128
7060 RETURN

7100 REM POINT PLOT
7110 IF X<0 OR X>19 THEN RETURN
7120 IF Y<0 OR Y>31 THEN RETURN
7130 POKE PEEK(16396)+256*PEEK(16397)+1+33*(19-X)+Y,XP
7140 RETURN

7200 REM DRAW A LINE
7210 IF ABS(X1-X0)> ABS(Y1-Y0) THEN GO TO 7300
7220 LET Y2=Y0
7230 IF Y0>Y1 THEN LET Y2=Y1
7240 LET Y3=ABS(Y1-Y0)
7250 FOR Y=Y2 TO Y2+Y3
7260 LET X=X0+(X1-X0)*(Y-Y2)/Y3
7270 GO SUB 7100
7280 NEXT Y
7290 RETURN
7300 LET X2=X0
7310 IF X0>X1 THEN LET X2=X1
7320 LET X3=ABS(X1-X0)
7330 FOR X=X2 TO X2+X3
7340 LET Y=Y0+(Y1-Y0)*(X-X2)/X3
7350 GO SUB 7100
7360 NEXT X
7370 RETURN

7400 REM PRINT X$
7410 IF X$="" THEN RETURN
7420 LET XP=CODE(X$)
7430 GO SUB 7100
7440 LET X$=TL$(X$)
7450 LET Y=Y+1
7460 GO TO 7410
```


Creating a Program

Doing your own thing

Although there is much pleasure to be had from running the many programs in this book, the most rewarding aspect of computing is the creation of your own programs. The feeling of satisfaction that can be obtained when, at last, your first vague ideas have been turned into a solid, working, program, can be immense. To ease the process a little for newcomers to the art, this section looks at some aspects of the creation of a good program.

Plan it

The first, and most important, point is that successful programs are not written in the way that one writes a letter to a friend; by starting 'Dear John ' and carrying on covering topics as they come to mind, with perhaps a p.s. at the end because you forgot to mention something in the body of the letter. Rather, they should be created as a work of art or a large building is created; by starting with a master sketch or plan and refining that until you are sure that the basic structure is correct before proceeding to the more detailed work. The temptation to actually enter some program lines into the computer should be resisted until you are sure that you have an exact plan of what you want to write.

The reason is that most programs include so much detail that it is extremely difficult to see what is happening by simply looking at the listing. It is rather like trying to understand the wiring of a car without a circuit diagram. If you try to understand the workings of someone else's program (or one of your own that you wrote a long time ago), you will soon find yourself having to sketch out some form of overall plan or flow chart.

The Flow Chart

Is a diagram - which should be as simple as possible - which shows the basic structure of the program, or part of the program. That is, it shows how the program progresses from one action to the next.

The structure of a BASIC program is made up of the following elementary structures;

- Sequential flow; the program progresses directly from one action to the next (to the next highest numbered program line).
- Unconditional Branch; the program jumps either forwards or backwards to another point in the sequence. E.g. GO TO 200
- Conditional Branch; the program may jump to another point in the sequence depending on the result of a comparison.
  E.g. IF A>B THEN GO TO 300
- Conditional Execution; a particular action is performed only if the result of a comparison is valid.
  E.g. IF A<0 THEN LET A=0
  or IF A=1 THEN GOSUB 1000
- The FOR - - NEXT loop.

The art of programming lies in developing flow charts which accurately represent exactly what you want the program to do, and which are as simple as possible. Thus the use of GO TO's are to be avoided wherever possible as they complicate the diagram (some theoreticians say that a good programming language shouldn't contain a GO TO type of statement !). The first flow chart to be drawn should show little more than the outline of the program; how it starts and finishes, and where the main calculation and printing are to be done. The details of how the calculation is to be performed, and any other complex parts of the program, should be left to a second 'level' of flow charts, which may in turn depend on third or deeper levels to show the fine detail.

Data

If your program will use more than just a few simple variables, then spend some time thinking about the best way of storing the data. The choice of how you represent data often has a crucial effect on the workings of your program. For example, if you want to manipulate characters within a string, it may be best to store it as an integer array (each element of the array being equal to the CODE of the corresponding character in the string). Also, if your program uses a lot of data and each data element can only have a small range of values, then it may be worth 'compressing' your data so that two or more elements can be stored as one integer variable - for examples see the ANIMALS and HEXPAWN programs in this book.

Subroutines

When creating a program you will often find that you need to include the same, or a similar, function more than once. By writing the function as a Subroutine, it need only be written once but can be called up as many times as required, thus saving both effort and memory space. For this reason it is usually worth while spending a bit of time during the program planning in looking at similar functions within your program to see if they can be dealt with by a suitable general purpose subroutine.

Subroutines may also be used to make the program listing easier to understand - and therefore less likely to contain errors. If a particular function occupies many lines of your program, it is worth writing that function as a subroutine even if it only occurs once within your program, as this will reduce the length - and hance the apparent complexity - of the main program. As an example, see the HORSE RACE program, where the subroutine starting at line 2000 is used only once.

Clarity Counts

The computer will do whatever you tell it - the problem lies in making sure that you tell it to do the right thing. This depends to a great extent on your understanding exactly how the program you are writing works. So take it in easy stages, starting with an overall plan, and gradually fill in the details in a methodical way, and don't resort to programming 'tricks' unless you really have to.

Other aids to clearer programs include;

- The use of REM statements to add explanatory notes in the program listing. Unfortunately they do take up valuable memory space, and for one's own use notes in the margin of a handwritten listing may be better.
- Making calculations and comparisons as explicit as possible, thus;

```
IF CODE(A$)=CODE("Y")
```

is more easily understood than;

```
IF CODE(A$)=62
```

which should only be used where the saving of a few memory bytes is crucial.

- The use of meaningful names for variables where possible, e.g;

```
LET LOSS=COST-PRICE
```

Again this uses a few more bytes of memory, but if you can spare them then why not ?

- Allocating a 'block' of lines numbers to each stage or subroutine in your program, so that - say - the various stages of your main program start with the line numbers 100, 200, - and the subroutines with 1000, 2000 etc.

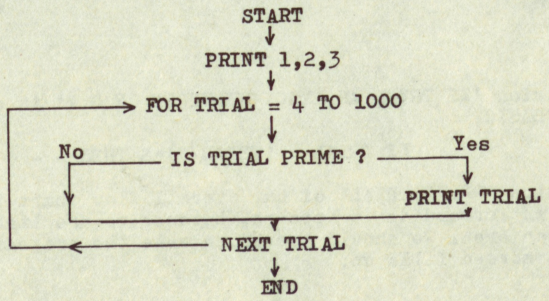
Primes ; an example

To illustrate the topics covered in the previous section, let's create a program which will calculate and display the Prime Numbers between 1 and 1000.

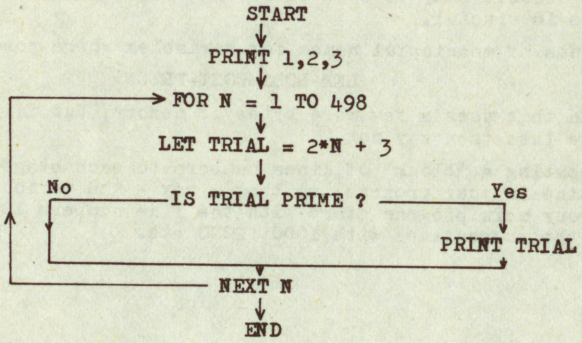
The program will use the brute-force strategy of looking at each of the possible numbers in turn and seeing if it is an exact multiple of any smaller number (other than 1); if so then it is not Prime and the program should move on to the next number. Our program will cheat a little by printing the first 3 Prime Numbers without calculation.

The first flow chart

Will not be concerned with the detailed calculation, but just the overall functions of starting the program, generating the sequence of numbers to be tried as possible Primes and printing those that turn out to be Prime, and ending;



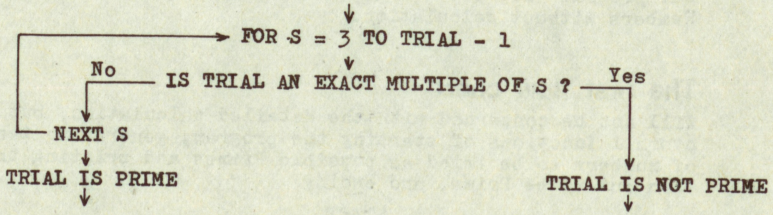
This raises two points. The first is that ZX80 4K BASIC does not allow a multi-character variable name such as TRIAL as the control variable in a FOR - - NEXT loop. The second is that since no even number can possible be Prime, we only need to generate odd numbers for TRIAL. We can overcome both of these problems by amending the flow chart to read;



Which is as detailed as we want to go for the first level of flow chart.

There are two areas in this flow chart ('IS TRIAL PRIME' and 'PRINT TRIAL') for which the coding is not immediately obvious, and which therefore need looking at in more detail.

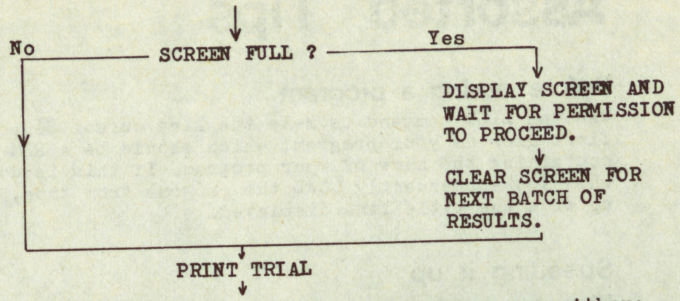
Taking the section 'IS TRIAL PRIME' first, this section has to see if TRIAL is an exact multiple of any number smaller than it (other than 1), and we can do this by generating a sequence of numbers (S, = 2,3,4,5 - -) and seeing if TRIAL is an exact multiple of any of them;



The decision 'IS TRIAL AN EXACT MULTIPLE OF S ?' is easily done in ZX80 BASIC;

```
IF TRIAL = (TRIAL/S)*S THEN - - -
```

The section 'PRINT TRIAL' of our overall flow chart is not quite as straightforward as it appears, because we are liable to overflow the screen area. We should therefore make the program pause each time the screen fills up;



To decide whether the screen is full, the program can either;

- Increment a variable each time a Prime is printed, and stop when the variable reaches a pre-determined limit. The variable will be re-set to zero when the screen is cleared ready for the next batch of results. Initially - before the first Prime is printed - the variable will have to be set to a starting value by the main program.
- Look at the content of memory location 16421. This is one of the ZX80's 'System Variables' and can tell us in which line of the screen the next character will be displayed; see page 123 of the Sinclair 'ZX80 Operating Manual' for details.

The second approach is slightly simpler, so the section of code becomes;

```
        IF PEEK (16421) > 3 THEN GO TO n
        PRINT "PRESS NEWLINE FOR MORE"
        INPUT A$
        CLS
n       PRINT TRIAL
```

The final program

We are now in a position to write our program;

```
  1 REM PRIMES
100 PRINT 1
110 PRINT 2
120 PRINT 3
200 FOR N=1 TO 498
210 LET TRIAL=2*N+3
300 FOR S=3 TO TRIAL-1
310 IF TRIAL=(TRIAL/S)*S THEN GO TO 500
320 NEXT S
400 IF PEEK(16421) > 3 THEN GO TO 440
410 PRINT "PRESS NEWLINE FOR MORE"
420 INPUT A$
430 CLS
440 PRINT TRIAL
500 NEXT N
```

The main program loop, which generates the sequence of values of TRIAL, is from lines 200 to 500. Within this loop, lines 300-320 see if TRIAL is Prime, and if so lines 400-440 print it out.

Assorted Tips

Before saving a program

Use the LIST command to make the line cursor ▶ point to the first line of your program, which should be a REM or a PRINT containing the name of your program. If this is done, then when you subsequently LOAD the program from tape, it will come up with the title line displayed.

Speeding it up

If your program takes an embarrassingly long time to run, examine the innermost loops - which are executed the most often - to see if there are any redundant operations or instructions which could be removed. For example, in the Primes program the inner loop steps S through all of the numbers from 3 to TRIAL-1, whereas a bit of thought will show that S really needs only to be stepped through the odd numbers from 3 to $\sqrt{\text{TRIAL}}$.

When a GO TO, GO SUB or NEXT is encountered, the ZX80 has to search through the program to find the line it has to jump to. As it starts the search from the first line in your program, loops nearer the beginning of the program listing will be executed faster than if they were at the end.

Similarly, every time a variable name is used, the ZX80 has to search through the program variable area to find it. Integer variables and Integer Arrays are stored in the order in which they are encountered during the execution of the program, so those which occur earliest in the program will be found first.

The times taken by the ZX80 to find the program line or variable are small by human standards, and normally you won't notice their effect on the time taken to run a program, but when you have a loop which may be executed thousands of times during the program, it may be worth putting the loop as near the start of your program listing as possible.

Breaking free

When you are trying out a new program, you often want to stop it running (to change something) when it is executing an INPUT statement. The problem here is that you have to type something, and if what you type doesn't have the correct syntax (if it isn't a proper integer or string expression) then the ZX80 won't accept it but will just sit there waiting for you to 'correct' your input !

The 'BREAK' key won't work either when the program is waiting for an input, as the ZX80 then treats this key as a space, and either accepts it (if it is expecting a string input) or gives a syntax error (if it is waiting for a numeric expression).

For example, run this;

```
10 FOR N=1 TO 1000
20 INPUT A$
30 NEXT N
```

then try to stop it, without turning the power off.

The answer is to input something which has the correct syntax, but which causes the ZX80 to give up for some other reason.

Thus, if it is expecting a string expression (as in the program given above), rub out the two quotation marks, and enter;

```
CHR$(99**99)
```

Alternatively, if the program is waiting for an integer input, enter;

```
99**99
```

In either case, the program will stop, and an error message of the form 6/n will be displayed, where n is the line number of the INPUT statement and '6/' means 'arithmetic overflow'.

Inadvertently entering EDIT (SHIFT/NEWLINE) in response to an INPUT will display one of your program lines ! Rub it out entirely then enter the correct input (plus the surrounding quotation marks if a string input is required).

Beware of lone REM's

A line containing only a REM - with nothing following it in that line - will cause the following program line to be ignored. Thus;

```
1 LET A=2
2 REM
3 LET A=3
4 PRINT A
```

will display '2'.

Debugging

Occasionally you may find that a new program doesn't run properly the first time. The process of finding out what is wrong and correcting it is known as 'debugging', on the basis that program faults are not caused by the programmer but by mischevous 'bugs' that must be discovered and rooted out.

The process is very much one of logical deduction, if possible tracing back from the fault to see what led up to it.

Techniques for doing this include;

- Adding extra PRINT statements at likely places, to let you know how the key variables are behaving - or which path the program is taking. (This is one reason for leaving gaps in the program line numbering sequence).
- Adding STOP statements at strategic points in the program. (Sophisticated programmers can make good use of an IF -- THEN STOP arranged to halt the program just before the error occurs.)

When a program has halted - for whatever reason - then the PRINT and LET statements used in immediate mode (without a line number) are very useful for examining and changing the values of variables. Having done this, a CONT will let the program carry on where it left off, or a GO TO will make it take some other course.

Converting other BASICs

A wealth of computer programs written in 'BASIC' can be found in other books and computing magazines, but as all versions of BASIC differ to some extent, it is unlikely that a program written to run on another computer will work on the ZX80 without some change. The extent and nature of these changes will depend greatly on the structure of the particular program and how it handles data, but it is possible to give some general guidance on things to look for when approaching the task of converting a 'foreign' program to run on the ZX80:

Multiple statement lines

Some BASICs allow multiple statements on a line, usually separated by : or \ , e.g;

```
10 LET A=B(2)+C : PRINT A,B,C
```

These will have to be written on separate lines for the ZX80.

Variables

Most BASICs allow floating point decimal numbers (generally called 'real' numbers) such as 9.73, -27.456.

Depending on the program it may be possible to 'scale' the values - by multiplying the values by say 10 or 100 for calculations so that a reasonable accuracy is maintained -, but be careful that the resulting values don't exceed 32767.

BASICs that allow real numbers will generally have a function INT(X) which gives the integer value of the real variable X. They may also use variable names of the form I% or A%. The % sign indicates that it is an integer variable.

Arrays

Are a critical feature in determining whether a program can be successfully translated to run on the ZX80. Some BASICs allow arrays with more than the 255 elements permitted on the ZX80, and many allow multi-dimensioned arrays such as;

```
DIM A(4,4,4)
```

which declares a three dimensioned array having a total of 125 (5x5x5) elements. This particular example could be converted by declaring a 125 element array

```
DIM A(124)
```

and calculating the required element in the course of the program. Thus

```
LET B=A(I,J,K)
```

would become

```
LET B=A(I + 5*J + 25*K)
```

Note that in some BASICs the first element of an array is A(1), rather than A(0)

LET

Some BASICs allow you to omit the LET word. Thus;

```
10 A = B
```

is the same as;

```
10 LET A = B
```

GOTO, GOSUB

Some BASICs do not allow a 'computed' GOTO or GOSUB such as

```
GO TO A+100
```

it may therefore be possible to simplify a program by taking advantage of this ZX80 facility

ON··GOTO, ON··GOSUB

Sometimes found in other BASICs, these statements make the program GOTO (or GOSUB) one of a number of lines, depending on the value of a variable. For example;

```
10 ON I GOTO 100,105,130
```

will jump to line 100 if I=1, 105 if I=2, and 130 if I=3.

Depending on the particular form used, they may be replaced by a computed GO TO or by a combination of IF.. THEN GO TO lines.

IF··THEN

Most BASICs will allow you to write

```
IF . . THEN 100
```

instead of

```
IF . . THEN GO TO 100
```

Some BASICs won't allow anything except a line number after the THEN. Programs written in these dialects may sometimes be simplified by taking advantage of the ZX80's ability to have a statement following the THEN. Thus;

```
10 IF X > 10 THEN GO TO 30
20 PRINT "TOO LOW"
30 - - -
```

could be re-written as;

```
10 IF X < 11 THEN PRINT "TOO LOW"
30 - - -
```

FOR··TO··STEP

Most BASICs allow the variable in a FOR - NEXT loop to be incremented by any value. Thus

```
FOR I = 2 TO 10 STEP 2
- - -
NEXT I
```

steps I through the even numbers from 2 to 10. This can be converted to run on the ZX80 by re-writing it along the

lines of;

```
FOR J = 1 TO 5
LET I = 2*J
- - -
NEXT J
```

INPUT

In some forms of BASIC, an INPUT statement can get more than one value - the elements being separated by commas. Also, an INPUT statement may contain a "prompt string", which is printed out before the INPUT is done. Thus;

```
10 INPUT "MONTH, YEAR", M,Y
```

would have to be re-written as;

```
10 PRINT "MONTH"
11 INPUT M
12 PRINT "YEAR"
13 INPUT Y
```

PRINT

PRINT statements will often have to be amended to make best use of the ZX80 screen format. Note that programs not written for the ZX80 will generally assume that print-out is to be done on a Teletype or a scrolling display, and may need changing to prevent a display area overflow.

TAB

Used in a PRINT statement, TAB(X) spaces to position number X on the print line. Page 124 of the ZX80 Operating Manual gives a routine to perform this function.

PEEK, POKE, USR

The effect of these commands depends entirely on the particular machine being used. To convert a program containing any of these to run on the ZX80 you must find out exactly what the command was intended to do, then write ZX80 code to perform a similar function.

DATA, READ, RESTORE

Most BASICs allow you to write a list of data elements in the program as;

```
100 DATA 4,9,6,273
```

When the program is run, a READ statement is then used to transfer the values to an array, e.g;

```
200 FOR I=1 to @
210 READ A(I)
220 NEXT I
```

which stores the value 4 in A(1), 9 in A(2) and son on. The RESTORE statement is used to go back to the beginning of the list if this is needed.

These functions can be replaced by a list of LET lines;

```
200 LET A(1)=4
210 LET A(2)=9
etc.
```


DEF, FN

A statement of the form

```
10 DEF FNA(X) = X + X*X
```

is acceptable to most BASICs, and defines a Function FNA(X), which in this example returns the value of X + X*X , and can be called up with different values during the course of the program, e.g;

```
20 LET A = 3 * FNA(2)
```

gives A the value 18.

Usually up to 26 functions ; FNA() to FNZ() are allowed.

To convert for the ZX80, we have to write the expression out in full each time it appears. Thus the above line would become;

```
20 LET A = 3 * (2 + 2*2)
```

END

Replace by STOP

Exponentiation

Some BASICs use the symbols ^ or ↑ to represent exponentiation; the ZX80 uses **

Arithmetic functions

The functions listed below are not available on the basic (4k ROM) ZX80, and if they occur in a program you wish to translate then some way must be found of calculating the correct value using the ZX80's arithmetic abilities;

```
ATN(X) : Arctangent of X
COS(X) : Cosine of X
EXP(X) : e raised to the Xth power
LOG(X) : log of X to the base e
SIN(X) : Sine of X
SQR(X) : Square root of X
TAN(X) : Tangent of X
```

The trigonometrical functions are expressed in radians.

RND

In most BASICs using real numbers, the expression RND(1) returns a random decimal fraction between 0 and 1, and is often scaled to produce a random number in the desired range. Thus;

```
10 LET A = INT(RND(1) * 100)
```

is equivalent to the ZX80 statement;

```
10 LET A = RND(100)
```

SGN

SGN(X) returns -1 if X is negative, 0 if X is zero, and 1 if X is positive.

String Arrays

The lack of string arrays (e.g. A$(I)) on the ZX80 may give rise to serious problems in translation. See the program 'Animals' for one possible way of overcoming this lack, or it may be possible to store the data as one long string and write routines to extract the required element (as in Buzz-Word), although it is not possible to change individual parts within the string.

Joining strings

Most BASICs will let you join two strings to form a third, thus;

```
10 LET A$ = "ABC"
20 LET B$ = "DEF"
30 LET C$ = A$ & B$
```

gives C$ the value "ABCDEF". (The + or ! symbols may sometimes be used instead of &). The lack of this facility on the ZX80 can prove awkward, as there is no easy way to perform the same function, except when printing;

```
10 LET A$ = "ABC"
20 LET B$ = "DEF"
30 PRINT A$;B$
```

giving a display of "ABCDEF".

ASC, CHR$

ASC(X$) returns the value of the first character in the string X$ and is similar to CODE(X$). Note, however, that while most computers use the ASCII code to represent characters, the ZX80 does not. Thus ASC("A") gives a value of 65 on most machines, but CODE("A") on the ZX80 gives a value of 38.

Apart from this difference in coding, the ZX80's CHR$(X) function behaves the same as in other BASICs.

VAL

Gives the numerical value of the string (of digits) X$, and is effectively the opposite of STR$(X). It may be replaced by a ZX80 routine, thus;

```
10 LET X = VAL(X$)
```

could be;

```
10 LET X = 0
11 LET A$ = X$
12 FOR I = 1 TO 5
13 IF A$ = "" THEN GO TO 20
14 LET X = X*10 + CODE(A$) - CODE("0")
15 LET A$ = TL$(A$)
16 NEXT I
20 - - -
```


LEN

Returns an integer value equal to the number of characters in the string X$. A ZX80 routine can be written to perform this function, thus

```
10 LET A = LEN(A$)
```

would be;

```
10 LET B$ = A$
11 FOR A = 0 TO 32767
12 IF B$ = "" THEN GO TO 15
13 LET B$ = TL$(B$)
14 NEXT A
15 - - -
```

LEFT$, MID$, RIGHT$

These functions operate on the string X$ and return a string which is a part of X$;

LEFT$(X$,Y) gives the first Y characters in X$.

RIGHT$(X$,Y) gives the last Y characters in X$.

MID$(X$,Y,Z) gives Z characters from X$, starting at position Y.

Depending on the particular application, it will usually be possible to write a special ZX80 routine to perform the same function. Thus;

```
LET A$ = LEFT$(B$,1)
```

becomes;

```
LET A$ = CHR$(CODE(B$))
```

and

```
LET A$ = MID$(B$,3,1)
```

could be written as;

```
LET C$ = B$
FOR I = 1 TO 2
LET C$ = TL$(C$)
NEXT I
LET A$ = CHR$(CODE(C$))
```

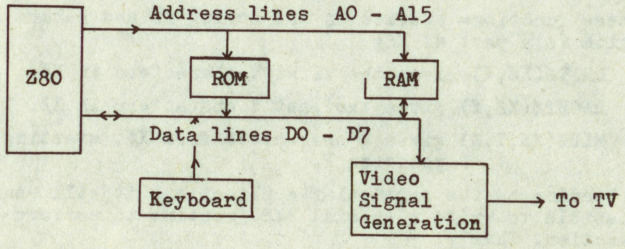
How it works

Overview

The main elements of the ZX80 circuit are;

- The Z80 Microprocessor itself (IC1).
- The Read Only Memory (IC2).
- The Random Access Memory (IC3,4).
- The Keyboard.
- The video signal generation circuitry.

They are linked together as shown in the simplified block diagram below;



The ROM and RAM each hold information organised as 8-bit 'bytes'. Each byte has an 'address', thus to read the value of a particular byte from ROM or RAM, the Z80 microprocessor first sets up the required address number on the lines A0 - A15, then issues a 'Read' command. The ROM or RAM (they respond to different groups of addresses) then places the required 8 bits of information onto the data lines D0 - D7 for the Z80 to read.

The information in the ROM has been permanently programmed in, and cannot be altered. It holds such things as the Basic Interpreter (of which more later), and the patterns for the displayed characters. The information stored in the RAM chips can be changed; to do this the Z80 puts the address of the RAM location to be altered onto lines A0 - A15, puts the data it wants to be stored there onto the data lines D0 - D7, then issues a 'Write' command. The contents of that RAM location will then be uptaded, and the old information previously stored at that address is lost. The RAM is used to hold temporary data such as your program, characters to be displayed, and the results of calculations. This information is lost when power is turned off.

The Memory Map given elsewhere in this book shows which addresses are used for ROM, and which for RAM. Your program may read the contents of ROM or RAM by using the PEEK command, and may use POKE to alter the contents of RAM.

The Z80 may also receive details of which key on the keyboard has been pressed, and data to be displayed on the TV screen is taken from the RAM by the video generation circuits at the appropriate times.

The BASIC Interpreter

The Z80 Microprocessor itself does not understand the BASIC language you enter your program in. Thus ' LET A = 2 + 3 ' is meaningless to it. What it will understand, and act on, is a much more detailed set of instructions known as 'Z80 Machine Language'. These are of the form 'Get the data byte stored in memory location 1234' or 'Take the next instruction from location 5678', and programming in this machine code is a time consuming and difficult task.

Therefore, so that you can program the ZX80 in a relatively easy language such as BASIC, the Z80 microprocessor runs a machine language program - stored in the ROM - known as the 'BASIC Interpreter'. When you are not actually running a BASIC program, the BASIC Interpreter program is continually sending the contents of the display file area of RAM to the TV, and scanning the keyboard. When it detects a keystroke, it receives the code for the key pressed, adds it to the display file and to a working space area of RAM (moving other information in RAM around if necessary - as for example if you had inserted a character in the middle of a line), moves the cursor, and checks the syntax of what you have entered. If you input is still incomplete (if you haven't yet pressed NEWLINE, or if there is a dreaded Syntax Error in the line), it then resumes the display and keyboard scan routines. However, if it is now in a position to do something - like running your program - the BASIC Interpreter then attempts to do as instructed (and stops generating a TV signal).

Your program is stored by the BASIC Interpreter in an area of RAM. When the RUN command is given, the Interpreter works its way through the program, taking each line in turn, and acting accordingly. For example, on reaching the program line;

```
10 LET A = 3 + 4
```

the Interpreter will cause the Z80 Microprocessor to;

- Note the current line number (10).
- Recognise the LET command.
- Find the RAM locations that have been used to hold the value of variable A. (If A hasn't been used before in the program then new RAM locations are allocated).
- Calculate the value of the expression on the right hand side of the = sign (3 + 4), and store the result in the RAM locations reserved for the variable A.

Similarly, on encountering;

```
50 GO TO 10
```

the Z80 will be made to search through your program for line 10, then continue from that point.

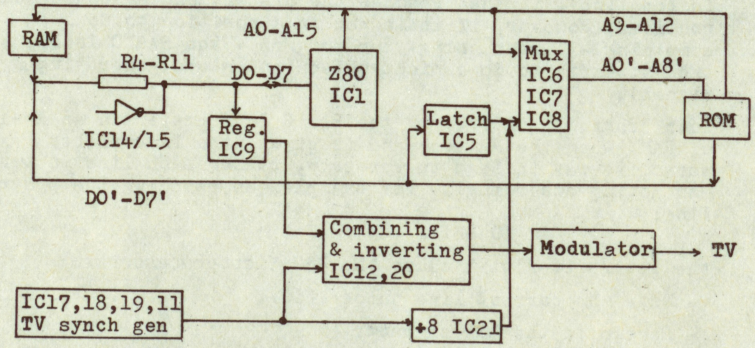
Binary Numbers

Numbers (e.g. the values of variables) are stored by the ZX80 in binary form - as this is easier for the machine to handle - and are translated to and from decimal form on input and output.

Each memory location contains 8 bits, giving $2^8 = 256$ possible combinations which are used to represent the numbers 0 to 255; this being the (decimal) range of values valid as data in a PEEK or POKE command. This range is, however, too limited to be of much use, and so the ZX80 stores the value of a variable in two adjacent memory locations, giving $2^{16} = 65336$ possible combinations, which are used to represent numbers in the range -32768 to +32767 (including zero). Memory addresses are similarly defined by 16 bits, but are taken as being positive numbers in the range 0 to 65335. One important point to note is that the first memory location of the two used to hold a 16 bit number (i.e. that with the lower address) is the least significant of the two. The program 'Decimal Peeker' gives an example of how to convert the contents of two memory locations into their decimal equivalent.

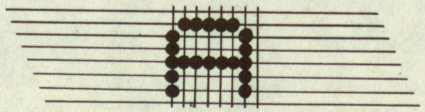
The Display

One of the clever features of the ZX80 is the way in which the TV display signal is generated with minimal additional hardware. The major circuit elements involved are shown below;



When the ZX80 is in the display mode, the Z80 Microprocessor IC1 controls the TV synch generation circuitry by its $\overline{IORQ}$, $\overline{RD}$ and $\overline{WR}$ outputs to produce a short line synchronisation pulse every 64uS - that is at the start of each TV horizontal scan line. Also, at 20mS intervals, a wide synchronising pulse is generated to lock the TV's vertical scan.

Each of the 24 lines of characters which can be displayed on the screen consists of 8 TV horizontal scan lines, and each character position on the screen is made up from 8 possible 'dots' in each of the 8 lines;



The dot patterns for the displayable characters are stored in the ROM. To display a line of characters the dot patterns for the top TV horizontal scan line are read out of the ROM in turn and sent to the TV. Then the dot patterns corresponding to the second horizontal TV scan line are read out, and so on until the line of characters is complete.

Each displayed character takes about 2.5uS of a horizontal TV scan line. During this time, the code for the next character to be displayed is read from the RAM and stored in the latch IC5. Then the dual port multiplexers IC6/7/8 are switched so that address lines A0' - A8' to the ROM are controlled by the outputs of the latch IC5 and the TV line counter IC21 instead of by the Z80 address lines A0 - A8. The ROM then gives out an 8 bit dot pattern which is then loaded into the 8-bit register IC9, from where it is shifted out one dot at a time, mixed with the TV line and frame synchronising signals, and fed via the UHF modulator to the TV set.

Although there are 128 possible display characters, the ROM only holds the dot patterns for 64; the other 64 are the 'inverse' characters formed by inverting the video signal in IC20 under control of the data bit D7'.

Display in Detail

While displaying a line of characters, the Z80 Microprocessor is executing a series of NOP (No Operation) instructions. Each of these instructions takes about 2.5uS (8 cycles of the 3.25MHz CPU clock),and consists of two parts;

- The 'Instruction Fetch' part, when the Z80 puts out the address of the memory location at which it expects the next machine language instruction to be stored, and reads that instruction from the data lines D0 - D7. During this time the open collector inverters IC14/15 are enabled, pulling D0 -D7 down to 0V, thus the Z80 sees a '00' (NOP) instruction, regardless of the actual content of the memory location accessed. This RAM location will in fact contain the code of the next character to be displayed, and this code is latched into IC5 as described earlier, the 1k resistors R4 - R11 preventing the actual RAM output lines D0' - D7' from being pulled to 0V by IC14/15.
- During the second part of the instruction cycle, the Z80 pulls its $\overline{RFSH}$ output low. This signal is used to switch the multiplexers IC6/7/8 as described previously. The Z80 also puts out an address which is incremented each instruction cycle, and was intended by the Z80 Microprocessor designers to be used as a refresh address for dynamic memories. In the ZX80, however, it is used to time each horizontal scan line, the starting address being chosen so that 64uS later A6 goes low, causing an interrupt which diverts the Z80 to another routine for generating a line synchronisation pulse and preparing for the next TV line scan.

To use RAM space efficiently, the ZX80 only stores in the display area of RAM the actual characters to be displayed. Empty lines and blanks at the ends of lines are not stored.

Each string of characters corresponding to a displayed line is terminated by the special 'End of Line' code 76. This code has two special properties;

- It is the only code which could reasonably be put in the display area of RAM that has bit 6 = 1.
- It is the Z80 Microprocessor HALT instruction.

When this code is read from the RAM, D6 being '1' turns off the open collector inverters IC14/15 via inverter IC13, gates IC16/17 etc. This allows the Z80 to 'see' a HALT instruction, which it obeys by executing a series of NOP instructions of its own accord until interrupted at the end of the horizontal TV scan line. While in this state, the Z80 HALT output is low, forcing the open collector inverters IC14/15 on throughout all of each instruction cycle, so for the remainder of the line the parallel to serial shift register IC9 is fed with the 'blank' 0's pattern.

Using USR

Together with the PEEK and POKE commands, USR lets us use routines written in Z80 machine code, rather than in BASIC. Apart from the sense of intellectual achievement that can be obtained by doing so, running programs in machine language can let us do things which are impossible in BASIC, or which will run much faster than their high level language versions.

To make use of USR, you need to be familiar with Z80 machine language. This is a complex subject and cannot be covered in this book, instead the reader is advised to study one of the Z80 books - such as "Programming The Z80" by R.Zaks - which are readily available from computer bookshops.

The machine language programs you write will need an area of RAM to operate in, which will not be overwritten by, or interfere with, a co-resident BASIC program. (Inevitably some form of BASIC program will be needed, if only to call up the machine language routine). There are several ways of allocating RAM space for this purpose, each having its own virtues and difficulties, and these are discussed in the following paragraphs. For examples of programs using USR, see MICROMON and MUSIC.

Using REM

If we make the first line of our BASIC program a REM statement, then we can use the RAM space occupyed by the characters following the REM to hold the machine language program. Thus if we enter;

```
1 REM ABCD
```


this will be stored at the start of the program area in RAM as (in decimal form);

location	value	
16424	0	} line number 1
16425	1	
16426	254	REM
16427	38	A
16428	39	B
16429	40	C
16430	41	D
16431	118	end of line

The locations 16427-16430 can now be changed (by POKE'ing) to hold the machine code we wish to run, as long as the ZX80 'End Of Line' code 118 (hex 76; the Z80's HALT instruction) is not entered. Unfortunately, once the REM statement has been altered in this way, displaying it (by listing that part of the BASIC program on the screen) is liable to make the system crash in peculiar ways depending on the actual values POKE'd.

Since the machine code is considered by the ZX80 to be part of a REM statement, it can be saved to and loaded from tape as part of the BASIC program.

Using DIM

By declaring an array with a DIM statement, then the variable space thus reserved can be used to hold machine language programs - with no restrictions on the codes which can be stored. Thus DIM A(255) will give a 512 byte space which can be accessed using PEEK and POKE or LET statements (in the latter case two bytes of machine code will have to be combined to form each element of the array).

The machine code will be saved on tape whenever the BASIC program containing the DIM statement is saved, and can be LOADed back - although the BASIC program should then be started with a GO TO n (where n is a line after the DIM statement) rather that RUN to preserve the machine code.

The only problem with this method arises because the RAM area used by the ZX80 to hold variables is not fixed - but 'floats' above the program area, so the machine language program can't use absolute addressing unless the associated BASIC program is not subject to any alteration.

In 'spare' RAM

The 'spare' area of RAM between the end of the screen display area (above DF-END) and the lower reach of the stack can be used to hold a machine language program, although care must be taken not to overlap the system stack or display areas. For this reason this technique is perhaps best suited to applications where there is enough spare RAM available to allow generous 'guard bands' around the machine

code. For those wishing to experiment, try putting your machine code as high in memory as possible but leaving about 50-100 bytes right at the top of available RAM for the stack.

Saving the machine code is not straightforward, but it can be done as for example by copying the code to a BASIC Integer array prior to SAVEing.

Above the stack

A BASIC program such as;

```
1 FOR I = 1 TO 100
2 GO SUB 3
3 NEXT I
```

will move the stack pointer down to leave a 200 byte space for machine code at the top of RAM.

Although this technique has the same problems in storing the machine code on tape as the previous one, it does offer the advantage that the machine code area is now protected from interference with the system stack and display area.

MICROMON

This is an extremely basic program which will allow you to enter and run machine language programs on a 1k byte ZX80.

When you run it, it first asks for an address, which should be entered as 4 hex digits. It then displays the contents of that location as 2 hex digits. You may then;

- Press **NEWLINE**. This will display the contents of the next memory location.
- Enter a 2 digit hex number, which will be stored in the current location. The program will then display the contents of the next memory location.
- Enter the letter G. This will run your machine language program by doing a USR to the current address. The machine language program should be terminated by a RET instruction (hex C9), which will return control to the BASIC MICROMON program and display the decimal value of the Z80's HL register pair.
- Enter any character which is not in the ranges 0-9 or A-G. The program will then ask for a new address. If you then just press NEWLINE without entering an address then MICROMON will stop (but note that using RUN to start it again will delete your machine code).

A 512 byte space for your machine language program is reserved by the Integer array A(), although MICROMON will in fact allow you to load machine code into any area of RAM. If the program is entered exactly as shown, this space will start at 417Eh. However, since this location will vary if for example you have included any extra spaces when entering MICROMON, it is best to check exactly where A() has been

stored by using MICROMON to find the (hex) values stored at locations 4008h and 4009h (system variable VARS). The first address of the space to be used to hold machine code is then the hex address obtained from these locations, plus 2. For example, with the program as written;

```
hex content of 4008h = 7C
 "      "    "  4009h = 41

combined address      = 417Ch

starting address of area for machine code = 417E
```

If MICROMON is saved on tape after a machine language program has been entered, then on subsequently re-loading it from tape, starting by a GO TO 10 (rather than RUN) will preserve the machine code.

As an example of using MICROMON, load the following machine code program;

addr	code	comment
417E	21	LD HL , 1234h
417F	34	
4180	12	
4181	C9	RET

Then run it (starting address = 417Eh). It should return with a display of 4660 (decimal equivalent of hex 1234).

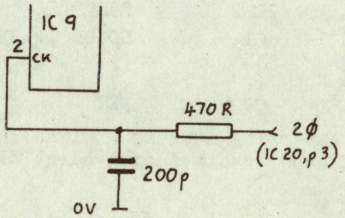
```
 1 REM MICROMON
 2 DIM A(255)
10 CLS
11 PRINT "ADDR"
12 INPUT A$
13 LET A=0
14 FOR L=1 TO 4
15 LET A=CODE(A$)-28+16*A
16 LET A$=TL$(A$)
17 NEXT L
20 CLS
21 LET P=A/256
22 GO SUB 50
23 LET P=A-P*256
24 GO SUB 50
25 PRINT " ";
26 LET P=PEEK(A)
27 GO SUB 50
28 LET A=A+1
30 INPUT A$
31 IF A$="" THEN GO TO 20
32 LET L=CODE(A$)
33 IF L < 28 OR L > 44 THEN GO TO 10
34 IF L=44 THEN GO TO 40
35 POKE A-1,16*L+CODE(TL$(A$))-476
36 GO TO 20
40 CLS
41 PRINT USR(A-1)
42 GO TO 11
50 LET L=P/16
51 PRINT CHR$(28+L);CHR$(28+P-L*16);
52 RETURN
```

Improving The Picture

If you 'fill in' an area of the screen by repeatedly writing one of the graphics characters (particularly the chequerboard pattern given by key A), then you may get unwanted vertical lines at the edges of the characters. To see this effect, run the following program;

```
10 FOR I=1 TO 320
20 PRINT "■";
30 NEXT I
```

The problem is caused by timing differences between the clock and load inputs to the shift register IC9. It can be alleviated by swapping R2 and C9; with 'C9' increased to around 200p as required by the particular computer.

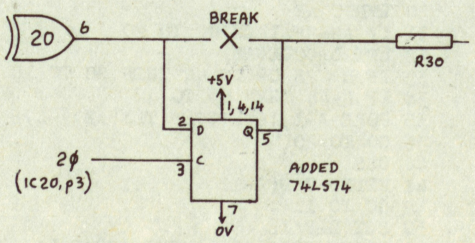


This still leaves another weakness in the video generation circuits, due to delays in the IC11,12,13 area. The effect is most noticeable if you have the ZX80 connected for white characters on a black background (board point A connected to B instead of C), and display a line of white squares. e.g;

```
10 FOR I=1 TO 10
20 PRINT CHR$(128);
30 NEXT I
```

If you run this program you will probably see a narrow vertical black line between each character position.

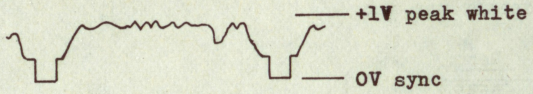
To overcome this problem it is necessary to add another IC - a 74LS74 - to re-time the video output signal.By cutting the 74LS74's leads short, and by laying it on its back, it is just possible to fit it between IC7 and IC12.



Connecting a Monitor

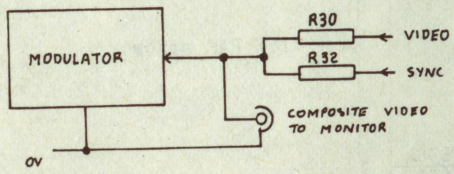
Although the ZX80 provides an adequate display once the TV set's tuning, contrast, and brilliance controls have been adjusted to their optimum positions, the definition of the picture is limited by the UHF modulation and de-modulation processes, and particularly by the receiver video filter.

Anyone lucky enough to own a TV monitor can get a much sharper picture by driving it with a straight video signal from the ZX80. Most monitors are designed to accept a composite (video plus sync) signal looking like;

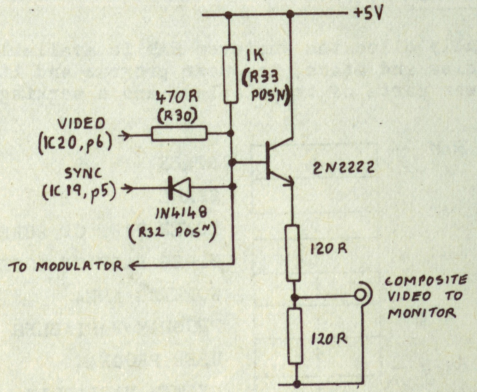


and generally the impedance at the video input connection will be around 75 ohms for feeding from co-axial cable.

In some cases it will be possible to get an acceptable result by connecting the monitor to the ZX80's modulator input;



However, for optimum results, the following circuit should be used. It involves changing R30 to 470 ohms, replacing R32 by a diode, and adding two other resistors and a transistor. They can all be made to fit in the case if care is taken. If the monitor doesn't provide a 75 ohm termination, then the lower 120 ohm resistor may be changed to 68 ohms. Note that this circuit still allows the ZX80's modulator to function correctly.



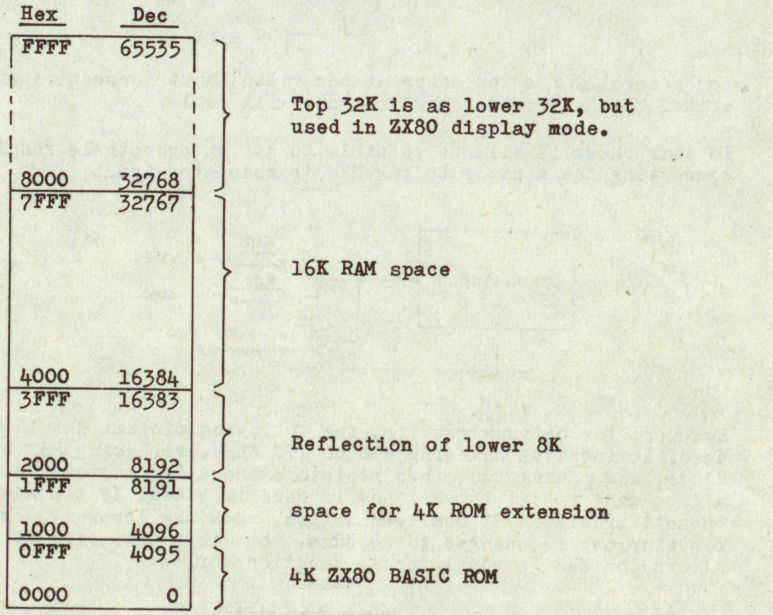
Memory Map

The ZX80's on-board RAM and ROM address decoding uses the address lines A0-A15 as follows;

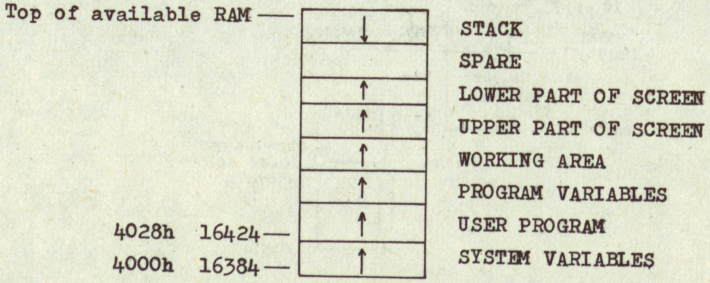
Z80 Address line ;	A15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ROM		0		0	A	A	A	A	A	A	A	A	A	A	A	A
RAM		1					A	A	A	A	A	A	A	A	A	A

Address line A15 is brought to a '1' when the ZX80 is in the display mode, otherwise it is a '0'. A13 is not used.

The resulting memory map is;



The ZX80 dynamically allocates whatever RAM is available between the system variables and stack, the user program and its variables, the upper and lower parts of the display, and a working space area;



Adding Memory & I/O

While you can run many interesting and useful programs using just the 1K on-board RAM, more memory is very useful especially if you want to use a lot of data in a program, or have a lot of output to be displayed.

Although all of the necessary signal lines have been brought out to the edge connector, adding any form of memory expansion to the ZX80 is a delicate business because of the 1K resistors R4-R11 in series with the Z80's data lines. These resistors greatly reduce the system's noise margins and prevent it from being able to drive any appreciable load (DC or AC).

Without changing the ZX80 itself, the only way of achieving a reliable memory extension is to connect only MOS loads to the data bus, and to keep the capacitance on the data lines as low as possible.

The Address and Control lines do not have this problem, and can happily drive two LS TTL loads connected to each line.

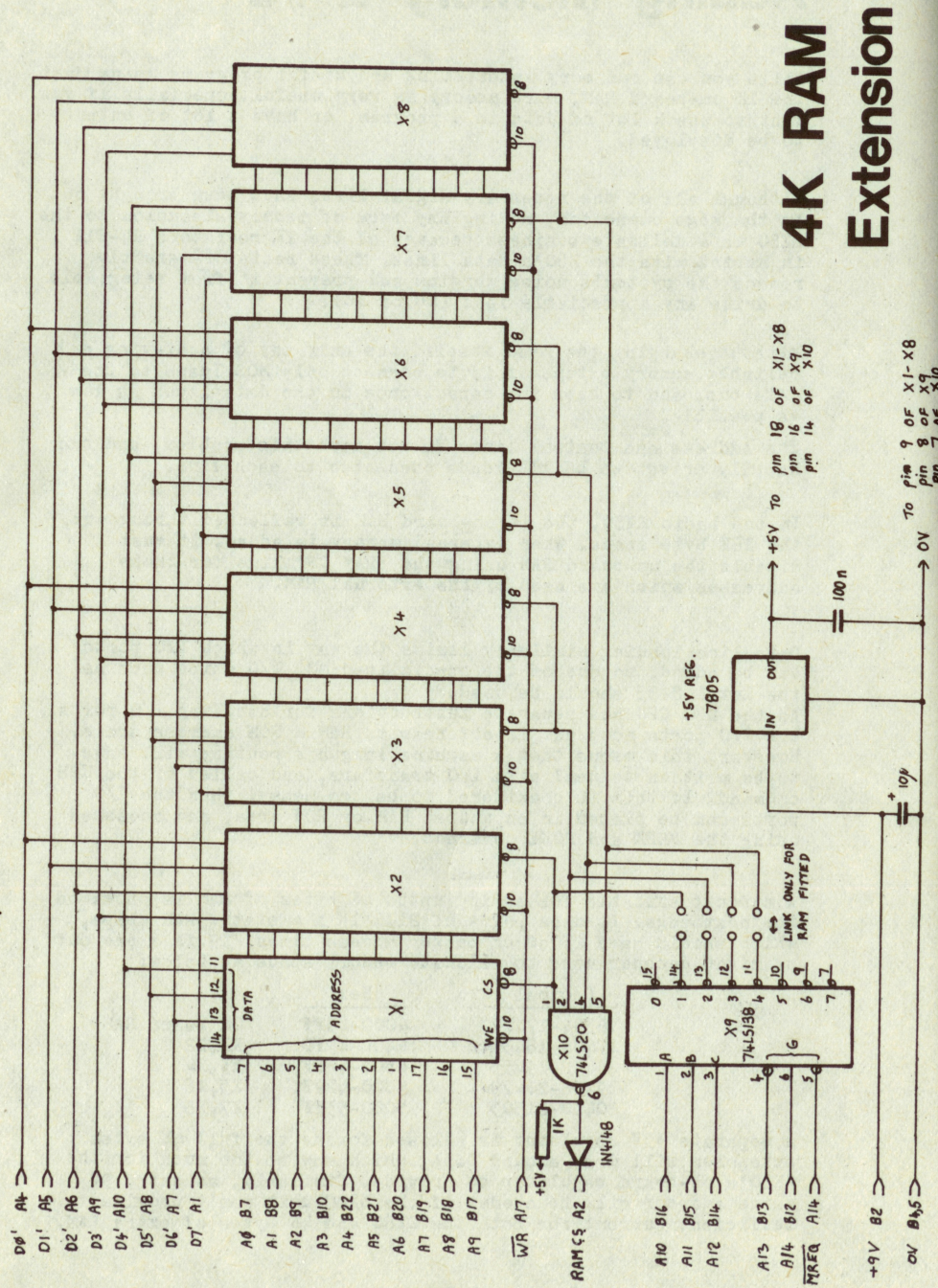
In the basic ZX80, the 1K on-board RAM is reflected throughout the 16K byte space. When external memory is added, it must disable the on-board RAM using the 'RAM CS' line for those addresses which are used by the external RAM.

Data line loading similarly limits the way in which I/O ports can be added; so one of the specialised MOS I/O chips such as the Intel 8255 should be used.
As the Z80 CPU has separate instructions for handling I/O ports, the I/O ports need not interfere with RAM & ROM address space. However, this means that a machine language routine will have to be written to deal with I/O transfers, and called by the USR command. If this is considered to be too messy, then the I/O ports can be placed in an unused RAM or ROM area, and accessed using the PEEK and POKE commands.

A circuit suitable for adding up to 4K bytes of RAM is shown on the next page. It uses pairs of 2114 1K x 4 static RAM chips, which should have 250nS or better access times. X9 is a one out of eight decoder used to allocate memory space as below;

Decimal	Hex	
16384-17407	4000-43FF	On board RAM
17408-18431	4400-47FF	X1,X2
18432-19455	4800-4BFF	X3,X4
19456-20479	4C00-4FFF	X5,X6
20480-21503	5000-53FF	X7,X8

A separate +5V regulator is allowed for as the full 4K bytes extension will draw around 0.5A, which may be too much for the ZX80's on-board regulator to provide. Similarly, an extra 9V mains adaptor will be needed if the ZX80 unit won't provide sufficient current for both the ZX80 and 4K bytes of extra RAM.



16K RAM & I/O Extension

For those wishing to use their ZX80 to the fullest possible extent, a circuit is given for an add-on board which will provide 16 K bytes of space for programs and data, and will give the user 24 programmable I/O lines to control music synthesisers, model train layouts, or whatever you wish.

It makes use of 8 16K x 1 dynamic RAM chips (X6-X13), an address multiplexer and refresh controller IC (X14), a Programmable Peripheral Interface IC (X15) , and 5 miscellaneous LSTTL IC's which perform the address decoding and some timing functions.

It is designed to plug directly into the rear ZX80 expansion connector, without requiring any modification to the ZX80. A separate power supply is needed to give at least the +12V and -12V lines; the +5V supply may be derived from the ZX80's internal regulator, alternatively the expansion board plus the ZX80 may be powered by a power supply such as Timedata's PZ100 which provides stabilised +5V, +12V and -12V lines ; reducing the dissipation within the ZX80's case. The power requirements of the extension board are; +5V @ 200mA, +12V @ 250mA, -12V @ 20mA. Note that if the +12V supply is connected without the -12V, then the memory IC's will overheat and could self-destruct.

The relevant memory addresses are;

RAM : 16384 to 32767 (decimal)
4000 to 7FFF (hex)

I/O :	decimal	hex	
	-16384	C000	Port A data
	-16383	C001	Port C data
	-16382	C002	Port B data
	-16381	C003	Control register.

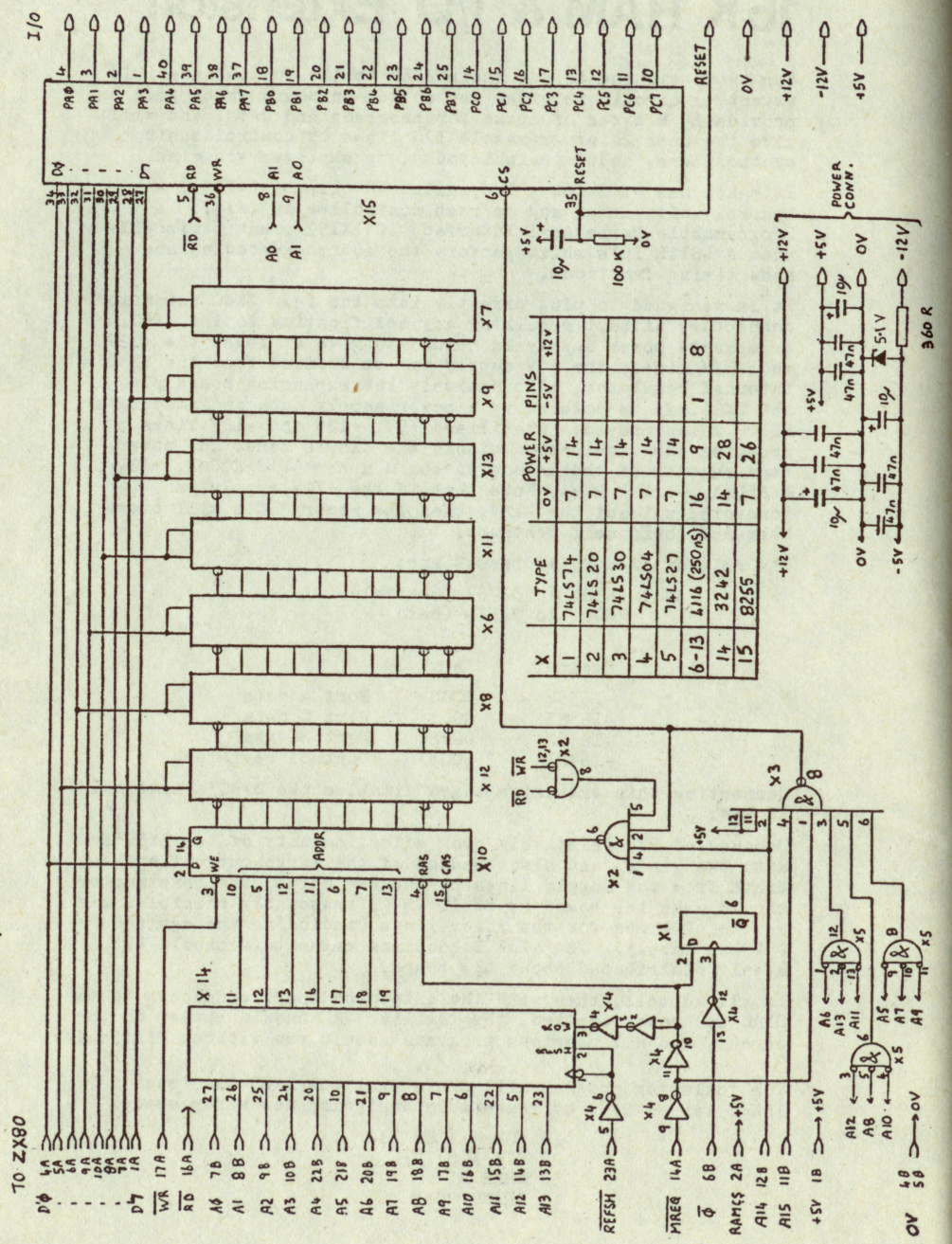
Connecting this extension board disables the ZX80's internal 1K RAM.

Because of the relatively poor noise immunity of the ZX80's data bus lines, and also because of the large current spikes drawn from the supply lines by the RAM chips, the constructor should make the board as small as is reasonably possible, and use thick lines for the power lines (including the derived -5V RAM supply). The 47nF decoupling capacitors should be evenly distributed about the board.

If all is well, then with the extension board connected to the ZX80 and power applied, the familiar K should appear on the screen, and all previous programs should run without difficulty.

The following program will prove that the ZX80 can 'see' 16K bytes of RAM, by repeatedly assigning new array space.

```
10 FOR N=1 TO 812
20 DIM A(8)
30 NEXT N
40 PRINT "OK"
```

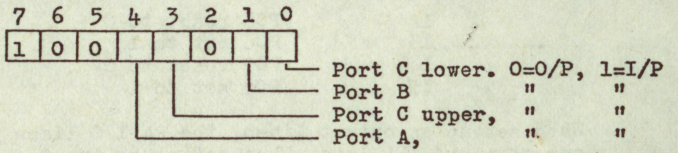


Using the I/O

This section summarises the main features of the 8255 Programmable Peripheral Interface chip, and gives enough information for most purposes. For more complex applications, refer to the 8255 data sheets

The 8255 has 3 8-bit ports (A, B and C) which may be used in Modes 0,1 or 2. Mode 0 is the simplest, and suitable for most applications.

In Mode 0, ports A and B may each be set up as 8 input lines or 8 output lines. Port C is divided into upper and lower 4-bit sections, each of which may be set up as input or output. Setting up the 8255 is done by writing a word to the Control Register (address -16381). The word to be written is constructed as;



Thus to set Port A for output, B for input, and both parts of Port C for input, we need to write an 8-bit pattern 10001011 to the Control Register.

Remembering that;

If bit 7 = 1 this corresponds to decimal 128
6 64
5 32
4 16
3 8
2 4
1 2
0 1

then we can set up the PPI by a BASIC statement;

```
POKE -16381,139
```

(139 = 128 + 8 + 2 + 1)

Having set up the 8255, we can now send a pattern to the output port A by;

```
POKE -16384,X
```

where X is the pattern (0-255)

The pattern on the 8 input lines connected to Port B can be read by;

```
LET A=PEEK(-16382)
```

All 8 bits of Port C can be controlled by;

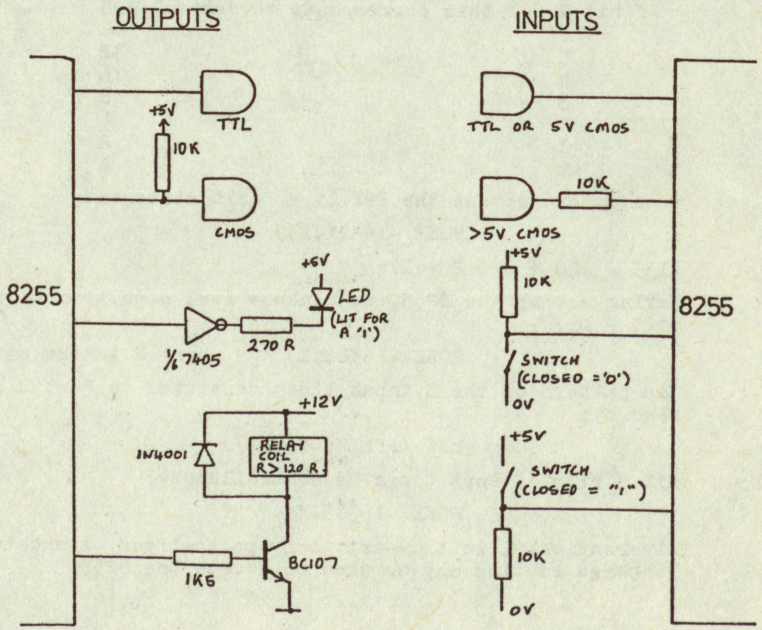
```
POKE -16383,X
```

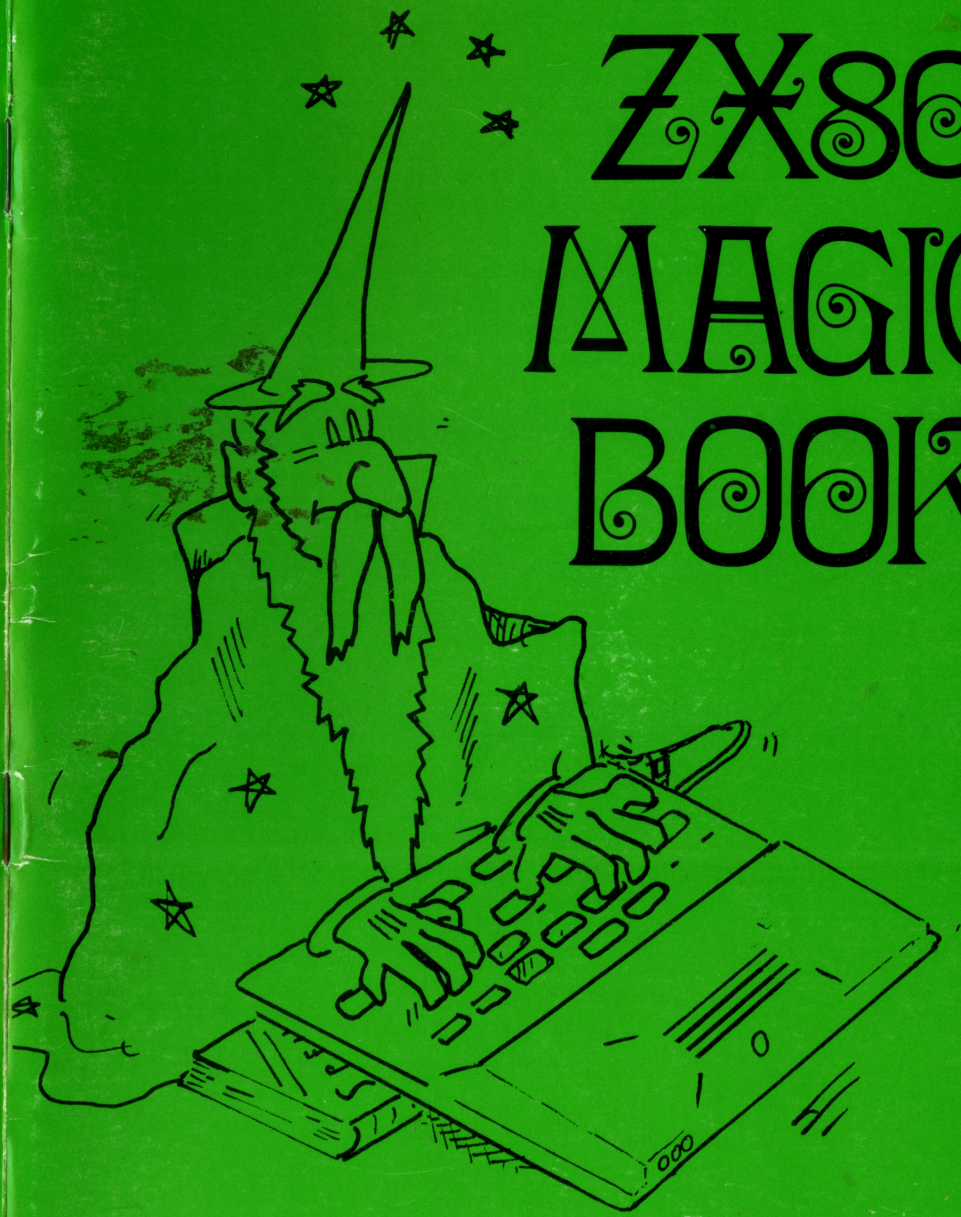
Alternatively, in time-critical applications, a machine language routine may be used to access the 8255.

The 8 Port C output lines may be set or reset individually at any time by writing one of the following values to the Control Register;

Value written	Effect
0	PC0 reset to 0
1	PC0 set to 1
2	PC1 reset to 0
3	PC1 set to 1
4	PC2 reset to 0
5	PC2 set to 1
6	PC3 reset to 0
7	PC3 set to 1
8	PC4 reset to 0
9	PC4 set to 1
10	PC5 reset to 0
11	PC5 set to 1
12	PC6 reset to 0
13	PC6 set to 1
14	PC7 reset to 0
15	PC7 set to 1

When set up as output lines, the 24 I/O lines can each drive one standard TTL load. When acting as inputs, they present a high impedance to the driving source, and accept an input voltage of less than 0.8V as a '0', a voltage between +2.0V and +5V as a '1'. Some typical interface circuits are given below;



WITH ZX81 SUPPLEMENT

THE ZX80 MAGIC BOOK